1 Introduction

This report investigates the performance of the Simulated Annealing, and Evolutionary Strategy optimisation algorithms on the minimisation of the *Rana's function*. For each algorithm a preliminary analysis is performed on the 2D *Rana's function*, to confirm that the basis components of the algorithm are working as desired. Following this a rigorous analysis of the algorithms application to the 5D *Rana's function* for a variety of different sub-methods and hyperparameter settings is investigated, with a focus on understanding the effects of the various components of the algorithm, as well as maximising performance. For both algorithms, simple variations to the standard components were proposed, and shown to improve performance. Finally an overall comparison is performed between the 2 algorithms, using random search as a baseline. All code was written from scratch, and is available in the Appendix.

2 The Problem: *Rana's function*

The optimisation problem this report deals with is the minimisation of *Rana's function* (Whitley et al. 1996), defined by

$$\begin{aligned} &\text{Minimize} \\ &f(\mathbf{x}) = \sum_{i=1}^{n-1} x_i \cos\left(\sqrt{|x_{i+1} + x_i + 1|}\right) \sin\left(\sqrt{|x_{i+1} - x_i + 1|}\right) + \\ &\qquad (1 + x_{i+1}) \cos\left(\sqrt{|x_{i+1} - x_i + 1|}\right) \sin\left(\sqrt{|x_{i+1} + x_i + 1|}\right) \\ &\text{subject to} \quad x_i \in [-500, 500] \text{ for } i = 1, \ldots, n \end{aligned} \tag{1}$$

Figure 1 shows the 2 dimensional form of *Rana's function*, showing that the problem has many local minima - making it a difficult optimisation problem. To find the best solution the following 2 elements are required: (1) a thorough exploration of the space, in order to find promising local minima, and (2) finding the lowest zone within a promising local minima. (1) and (2) have to be balanced, as they require the algorithm to perform different functions, where (1) requires testing many solutions far apart in control parameter space, and (2) requires a focuses search within a narrow zone of control parameter space. This trade-off is referred to as exploration-exploitation trade-off, and is a common theme throughout this report. Because *Rana's function* has an especially large number of local minima, it requires an especially strong emphasis on exploration.

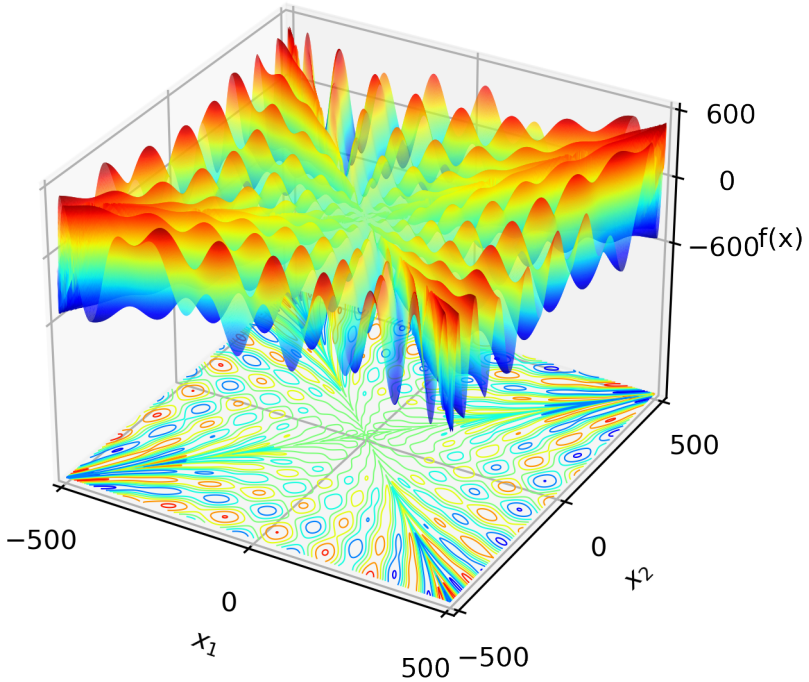

Figure 1: *Rana's function* 2D

3 Simulated Annealing

3.1 Implementation details

For each component of the Simulated Annealing algorithm the following methods were implemented, noting any deviations from the lecture slides by inserting a "⋆" at the start of where they are mentioned.

Annealing schedule: Simple exponential cooling, Kirkpatrick initialisation (Kirkpatrick 1984)
Control variable perturbation methods: (1) Constant spherical noise (`Simple-step`), (2) full adaptive covariance step matrix (referred to as `Cholesky-step`, as the Cholesky decomposition is utilised),(3) adaptive diagonal covariance step matrix (`Diagonal-step`).
⋆ Both adaptive control variable perturbation methods are prevented from having a determinant below 10^{-16}, and each element of the step size matrix was clipped to have it's absolute value within 10% of the control parameter range, to prevent the step size from becoming too small or to large in any direction, making the program more stable.
⋆ Significant experimentation with non-standard rules for **when** the updates to the step size control matrix occur was performed - as described in Section 3.3.2.
Convergence criteria: Absolute difference in accepted objective function $< 10^{-6}$ for last last 1000 objective function evaluations. Maximum 10,000 function evaluation.
Bound enforcement: Repeated sampling of perturbation until control variable within bounds.

3.2 Model Baseline on 2D Rana function

To demonstrate that the Simulated Annealing algorithm is working as desired: (1) the basis components of the algorithm (solution generation, solution acceptance, temperature scheduling) need to be shown to be working, (2) the algorithm has to demonstrate both exploration of the space, and exploitation of the most promising local zones of the space (exploration-exploitation trade-off).

For simulated annealing to work as desired, in the initial iterations the algorithm needs to focus on exploring as much of the space as possible, while towards the end of run, instead performing a focused search of the best "zone". This is performed primarily by the temperature scheduling, which is shown in Figure 2 along with the corresponding probability of acceptances for generated solutions, over the course of a single run of the program, using the `Simple-step` method[1]. As the temperature decreases, the probability of accepting solutions that increase the objective function becomes lower - focusing the optimisation on a narrower band of space. The effect of the temperature on the accepted solutions is shown in Figure 3, where the variance in the accepted objection function values is initially high - with many acceptances of objective values that were higher than the previous iteration. As the iterations continue and the temperature is annealed, both the running mean and variance of the accepted objective values generally decreases until convergence is reached. For the `Simple-step` method the step size remains fixed throughout the run, as is shown by the constant width of generated function evaluations in Figure 3. Together Figure 2 and 3 confirm that the basic mechanisms of solution generation, acceptance and temperature scheduling are working correctly together. Finally, the plot of the search pattern for this run (Figure 4) shows accepted solutions are initially distributed throughout the space, and then later on are narrowly clumped on local minima - showing that the algorithm is correctly balancing both exploration and exploitation.

[1] For all initial models, a hyperparameter setting of markov-chain length = 50, annealing alpha = 0.95 is used

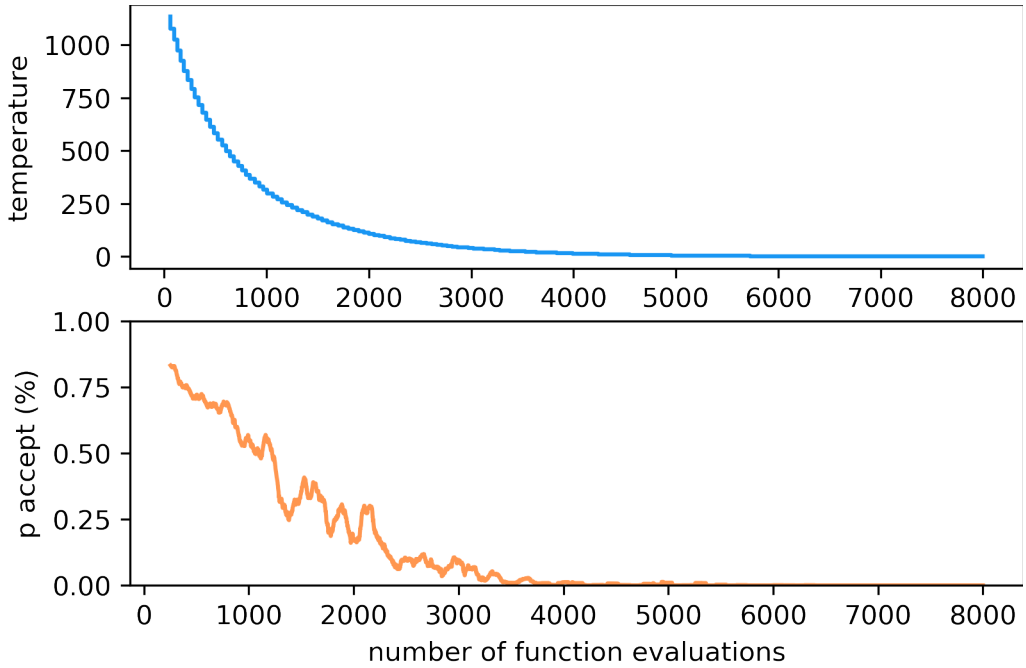

Figure 2: Temperature annealing and probability of acceptance for an example run of the Simulated Annealing algorithm using the `Simple-step` method.

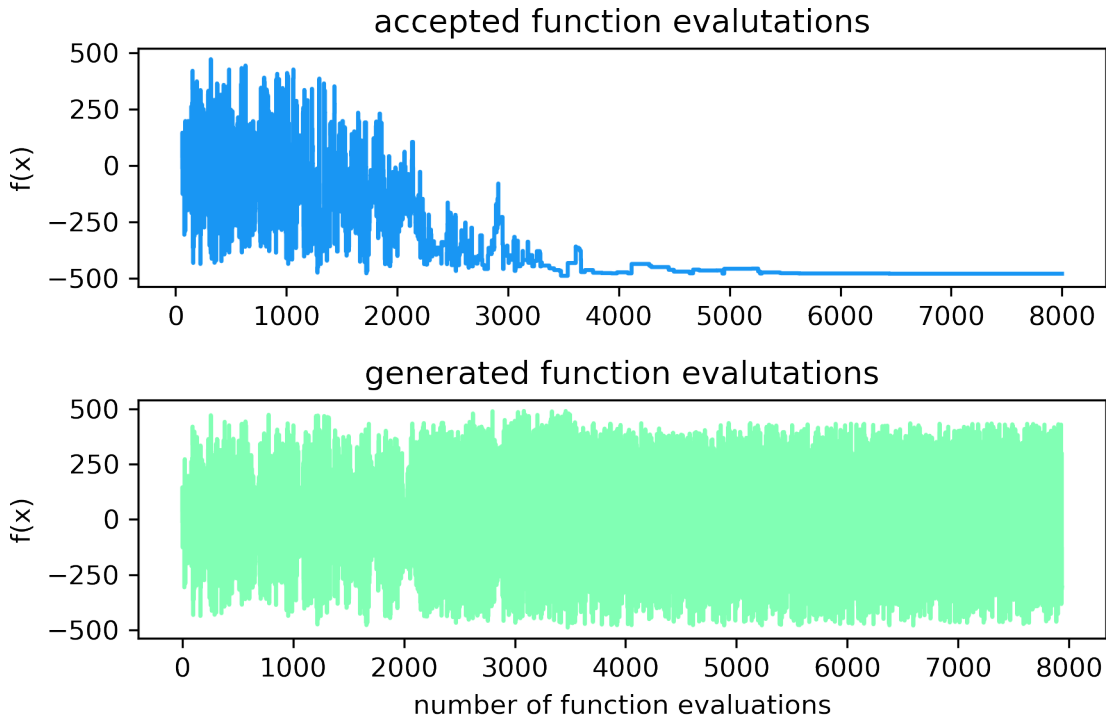

Figure 3: Accepted and generated function evaluations for an example run of the Simulated Annealing algorithm using the `Simple-step` method.

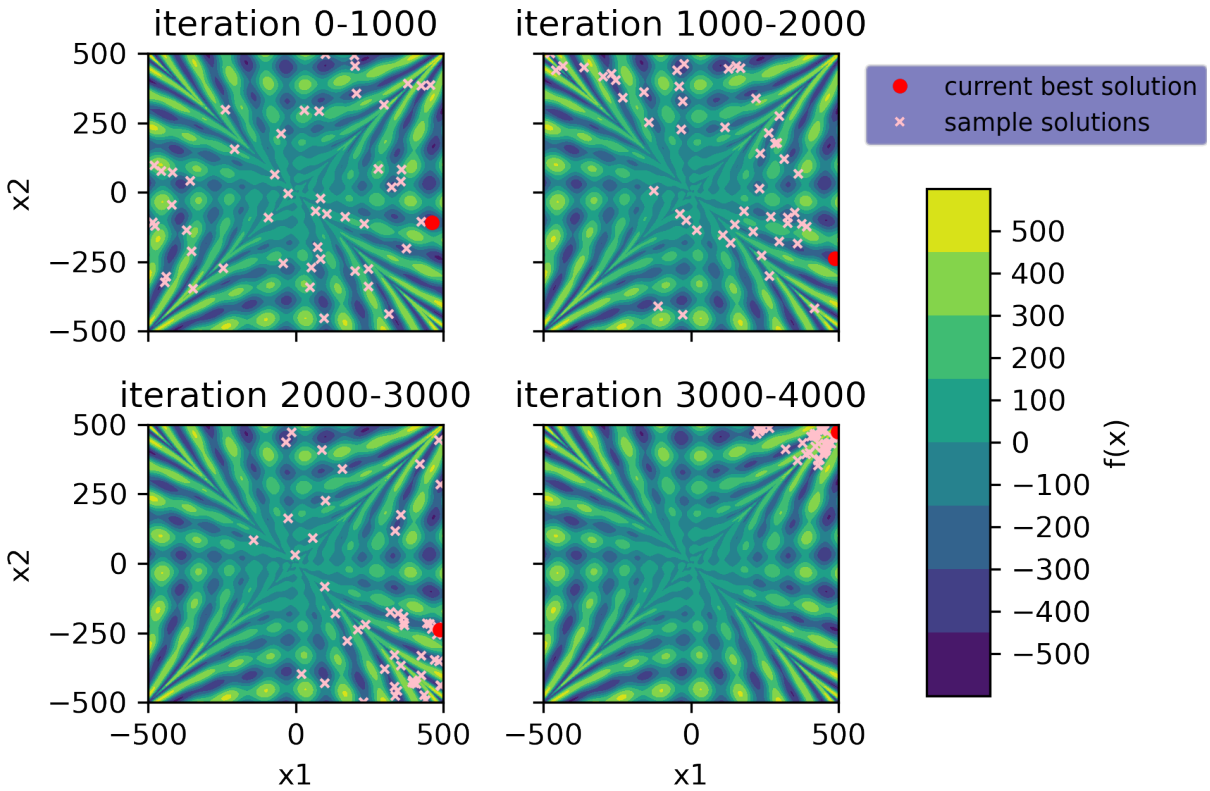

Figure 4: Sample solutions throughout optimisation for an example run of the Simulated Annealing algorithm using the `Simple-step` method. Early in the optimisation the whole space is explored. As the optimisation progresses, a progressively smaller subset of the space is focused on

The search patterns for example runs of the Simulated Annealing algorithm with the adaptive step size covariance methods (using the same seed, and temperature schedule) offer a clear visualisation of the effects of these methods, confirming that they are working as desired. Figure 5 shows the accepted solutions throughout the course of a run using the `Diagonal-step` method. Early in the run, the the standard deviations of each element are high, and the temperature is high, so the accepted solutions are scattered throughout the space. The adaptability of the matrix allowing for different variance for different elements of x is then demonstrated later in the run (bottom left hand plot), where the accepted solutions have lower variance with respect to x2 elements relative to x1. The example run of the `Diagonal-step` method also converges far faster than the `Simple-step` method's example run - by 2000 iterations there is virtually no variance of the accepted solutions (while with the `Simple-step` method there was still significant variance at 4000 iterations). This demonstrates ability of the `Diagonal-step` method to greatly reduce the overall step size throughout the optimisation, narrowing the space of **proposed** solutions (not just narrowing the space of accepted solutions via temperature cooling), allowing for faster convergence.

Figure 6 shows the progression of accepted solutions throughout the course of a run of the Simulated Annealing algorithm with the `Cholesky-step` method. Early in the run the additional noise from the step size covariance matrix is large and the temperature is high so the accepted solutions are scattered throughout the space. Later in the run, the accepted solutions are confined in a narrow space with a very high correlation between each of the elements of x (in the bottom left hand plot of Figure 6, the solutions appear in diagonal line). The narrowness of the space later on in the run is a function of both the lower temperature, and the lower overall noise of the sampling (from the determinant of covariance matrix decreasing). The high correlation between elements of x shows that the off-diagonal elements of the covariance in the `Cholesky-step` method are playing a significant role in the search pattern - contrasting the pattern from the `Diagonal-step` method.

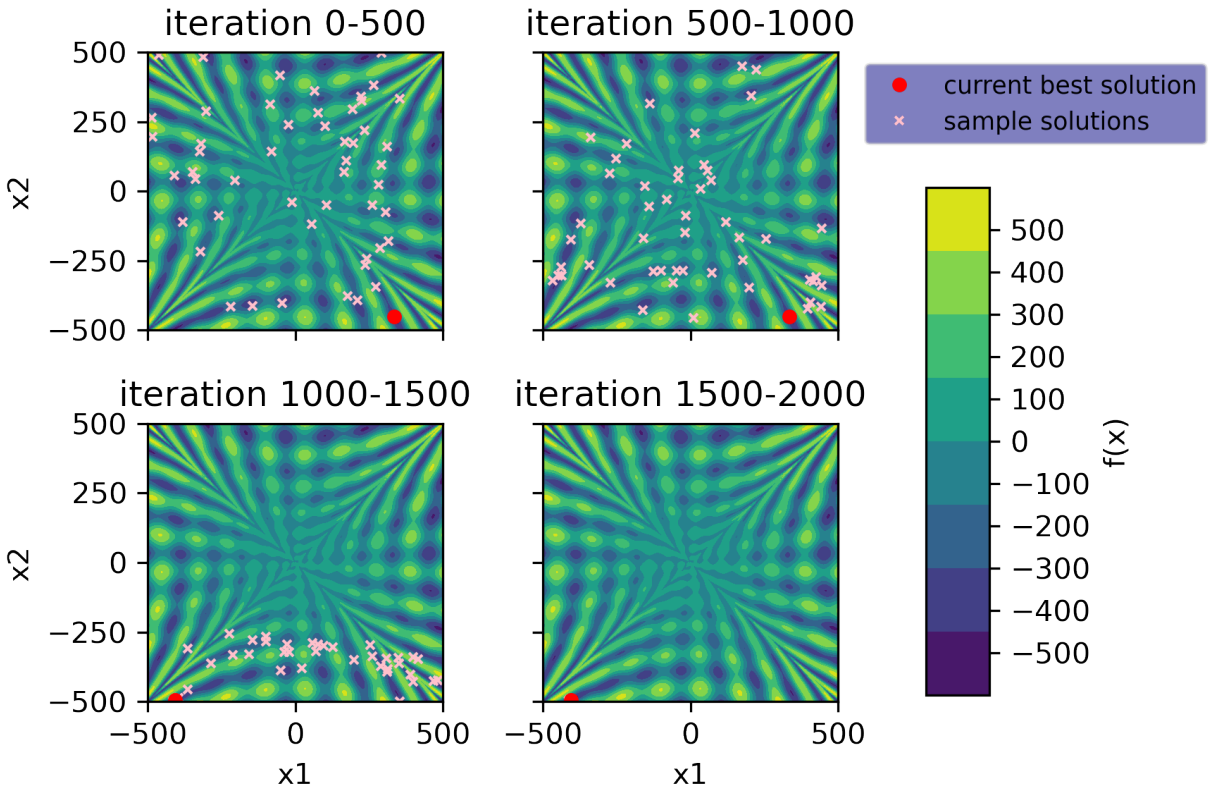

Figure 5: Sample solutions throughout optimisation for an example run of the Simulated Annealing algorithm using the `Diagonal-step` method. The difference in variance in the x2 and x1 elements is visible in the bottom left hand plot, where the accepted solutions have lower variance with respect to x2 elements relative to x1

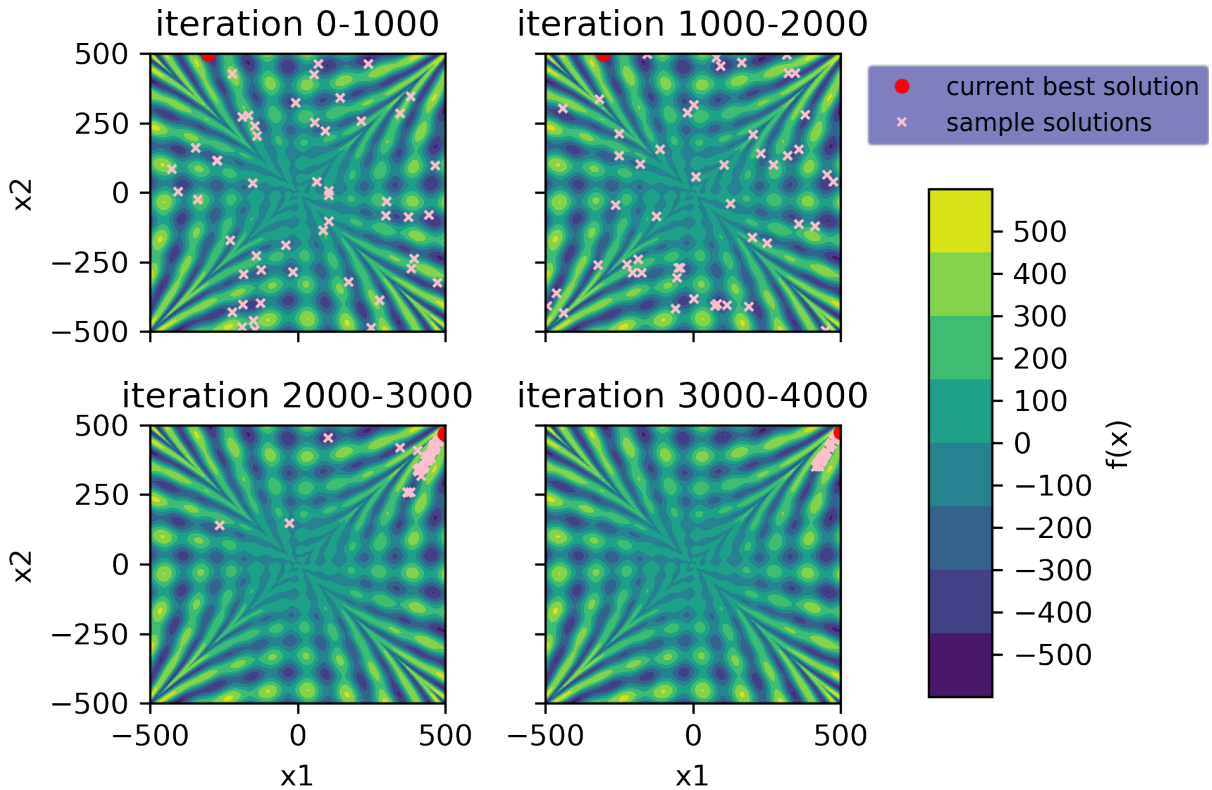

Figure 6: Sample solutions throughout optimisation for `Cholesky-step` method. The two bottom plots show significant correlation between x1 and x2 elements - demonstrating that the off-diagonal elements of the covariance matrix are having an effect on the search.

3.3 Analysis on 5D Rana function

In this section of the report, the effects of (1) each of the step-methods (`Simple`, `Diagonal`, `Cholesky`), (2) the step-updating rule (a proposed addition to the Simulated Annealing algorithm) and (3) the temperature annealing schedule hyperparameters (maximum markov chain length, alpha) are investigated. Each effect (1,2,3) is initially discussed and analysed separately. Because these changes to the algorithm have interacting effects, they cannot be optimised individually. Therefore a simple grid optimisation of combinations of each method and hyperparameters is performed. Finally, using the tuned hyperparameters, the overall performance of the Simulated Annealing algorithm, with the different step-size-updating rule and step-methods are compared.

3.3.1 Step Method

Various sub-plots describing the performance of the `Diagonal-step` method for an example run are given in Figure 7. The `Diagonal-step` method allows the standard deviations along each element of the control parameters to adapt over the course of the run - allowing (1) the algorithm to focus it's search along some dimensions more than others and (2) the overall variance to decrease, causing the generated solutions to focus on a smaller local zone. In Figure 7, the standard deviations along each element of x fluctuate throughout the run, and correspondingly the variance in the generated function evaluations visibly varies (e.g. the standard deviation with respect to the element corresponding to the purple line increases, causing the width of the generated function evaluations to increase). By 4,000 iterations, the overall variance has decreased significantly (all of the standard deviations become very small), causing the generated objective functions to greatly decrease in variance, eventually causing the algorithm to converge the before the 10,000 maximum function evaluation count is reached. The `Diagonal-step` method has an additional advantage, which is in that it allows for the acceptance probability to be adjusted to include division by the actual step size, which scales the probability of accepting perturbations according to how large the step size was (with large step sizes having increased probability of acceptance) - thus encouraging exploration.

Various sub-plots describing the performance of the `Cholesky-step` method for a single example run, are shown in Figure 8 below. To demonstrate the effects of having a full covariance matrix (as opposed to the `Diagonal-step method`): (1) the magnitude of the eigenvalues of the covariance are plotted - indicating the magnitude of the variance along each eigenvector of the covariance matrix and (2) the minimum angle between the eigenvector corresponding to the largest eigenvalue and each of the 5 axes is plotted in order to give an indication of how much covariance between different elements of the control parameters there is.
The covariance's eigenvectors decrease throughout the program, lowering the amount of variance in x, and correspondingly in the generated objective function values (Figure 8 subplot 2 and 3). The minimum angle between the axes, and the eigenvector corresponding to the largest eigenvalue is significant throughout the course of the optimisation - this indicates that there is a significant correlation between different elements of x (i.e the non-diagonal elements of the covariance are having an effect). The `Cholesky-step` method has the advantage of potentially being able to capture a richer description of the local topology, which would allow for a more efficient "explotation" of the local zone.

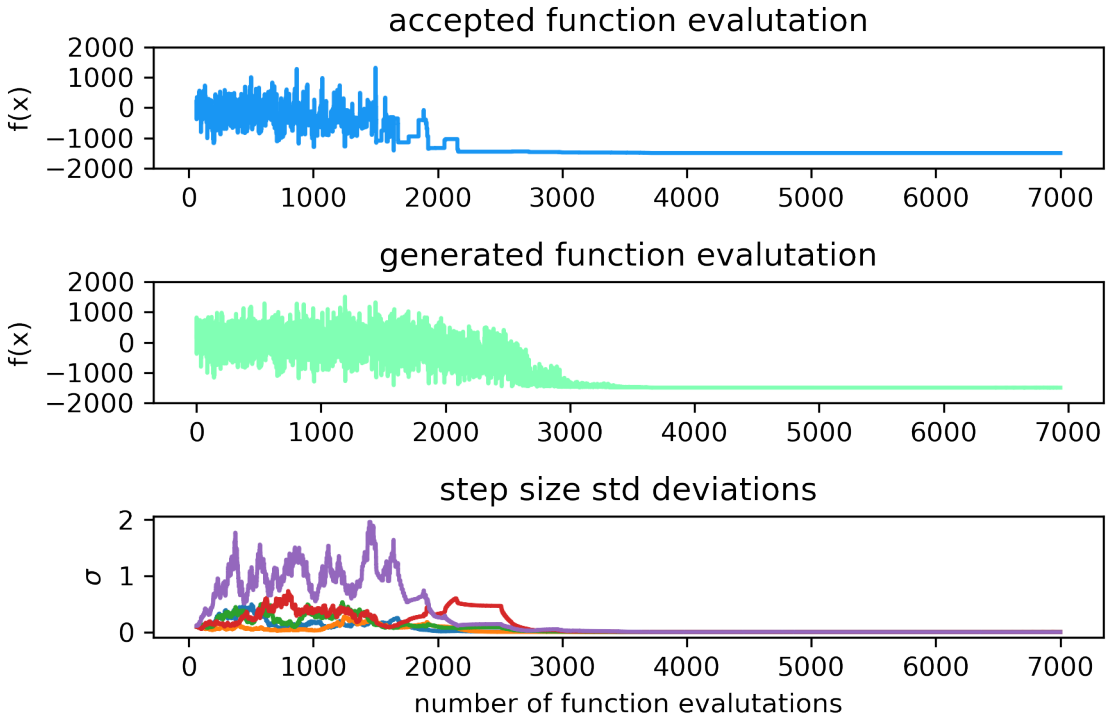

Figure 7: `Diagonal-step` method example run performance. Fluctuations in the purple line cause an initial increase in the variance of the generated objective functions. The decrease in all of the standard deviations by 3000 iterations, causes the generated function evaluations to greatly decrease in variance, eventually leading in early convergence (before 10,000 iterations).

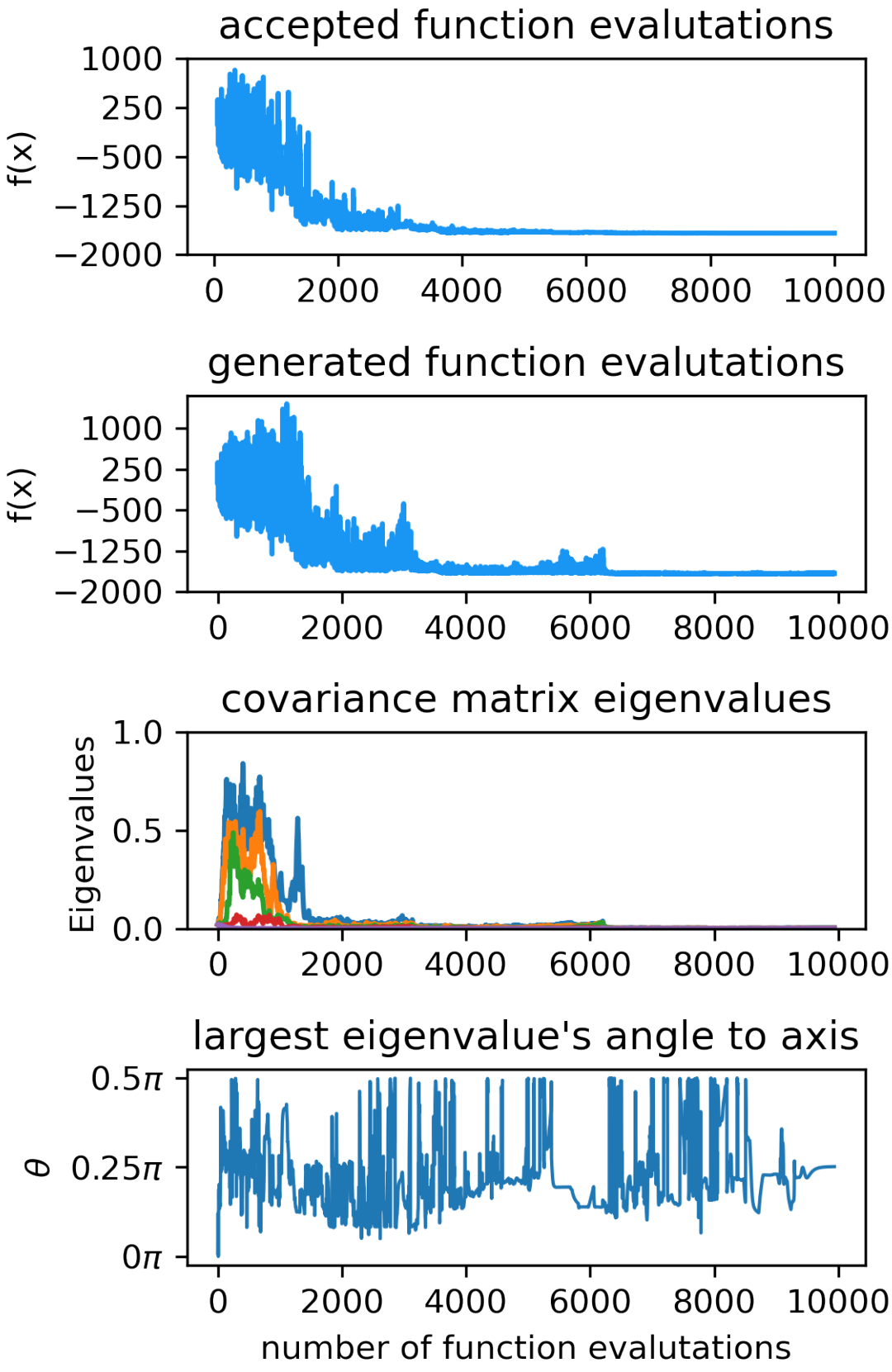

Figure 8: `Cholesky-step` method example performance. The magnitude of the variance in the generated objective functions has a clear link to the magnitude of the covariance eigenvectors (a decrease in the latter causing a decrease in the former). The largest eigenvalue's minimum angle to an axis is significant throughout the run, indicating that the non-diagonal elements of the covariance matrix are significant.

3.3.2 Step updating rule

For both of the adaptive step size methods the following issue was often observed: If the step size[2] is only updated when new values are accepted (as specified in the lecture slides, as well as in Busetti 2003), then if a reasonably good solution is found while the temperature is relatively cool and the step size is relatively large, then new solutions are almost always rejected (as the step size is too big) but because the step size is only updated when new solutions are accepted, the step size doesn't get updated, causing the program to get stuck at the current solution. I.e. there is a catch 22 where new solutions aren't accepted because the step size is too big, and the step size is not updated as there are no new solutions being accepted. An example of such a situation is shown in Figure 9, where after 2500 updates, for a good new solution to be discovered (such that it is accepted), the step size needs to be reduced, however because there are no new solutions that are good enough are discovered (because the step size is too big), the step size remains fixed and there is no improvement for the rest of the run.

This issue can be dealt with by instead updating the step size every time a perturbation is accepted **and** and at an interval (e.g. every 5 steps) independent to whether the perturbation is accepted. The reason for updating the step size is to adjust the covariance matrix to better fit the local topology - and this addition to the algorithm allows for information on the local topology to be folded into the step size matrix at a rate corresponding to both the number of acceptances and the number of total steps - preventing over dependence on the rate of acceptances for the step size to match the local topology. This is referred to as the `diversified-step-update-rule` for the rest of the report.

The effect of the size of the interval between step updates using the proposed rule in shown in Figure 10. For an interval of 10,000, this method becomes identical to the original step-size update rule of only updating the step-size-matrix after acceptances. Figure 10 therefore shows that the original rule has inferior performance and that the proposed rule for updating the step size has clear benefits. A step-update-interval of 4 and 45 are preferred for the `Diagonal-step` and `Cholesky-step` methods respectively using an initial configuration for the other hyperparameter settings[3][4].

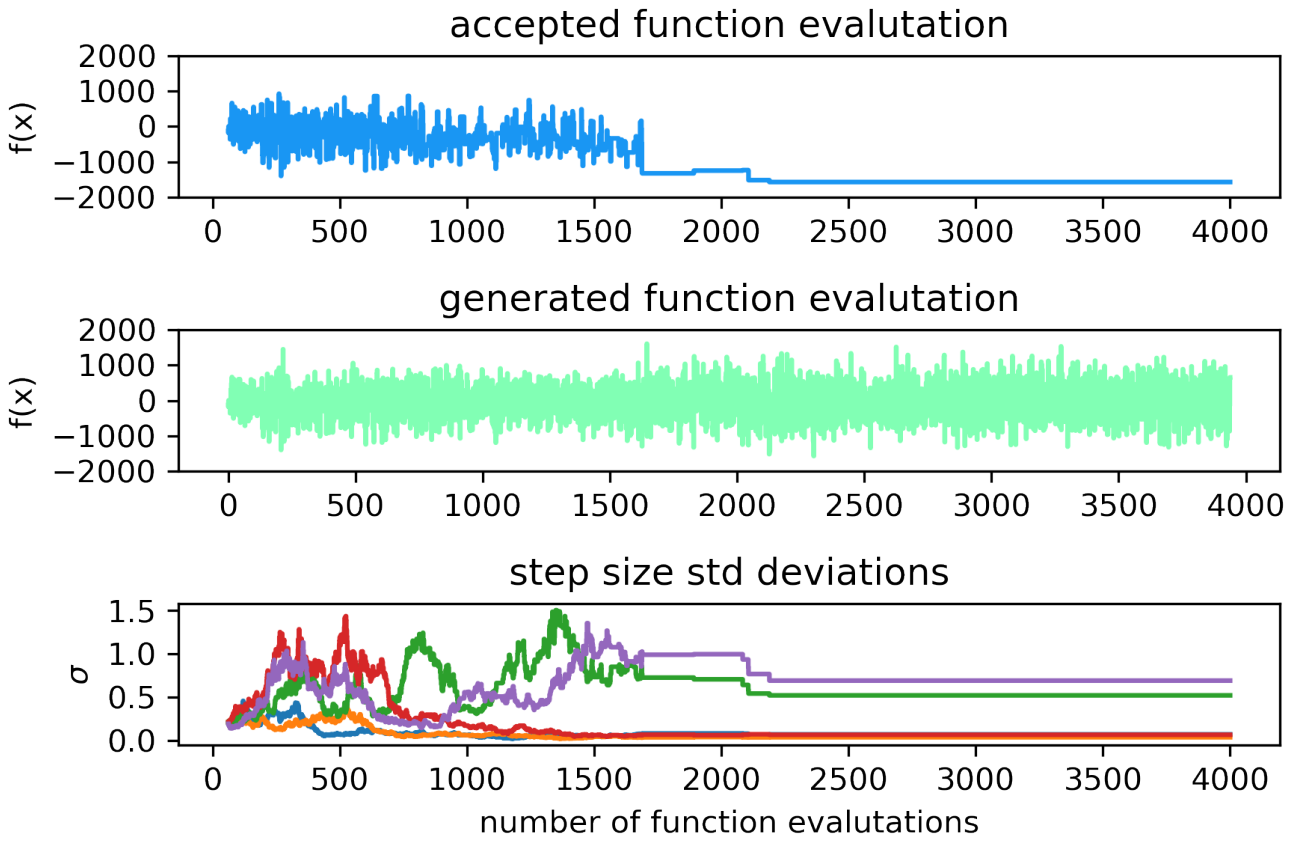

Figure 9: Illustration of issue where algorithm gets stuck, due to dependency on acceptances for step size updates.

[2]Here "step size" updating refers to updating the covariance matrix that controls the step size
[3]hyperparameter setting of markov-chain length = 50, annealing alpha = 0.95
[4]ultimately this is optimised by the grid search, so the specific results shown in Figure 10 are for illustration

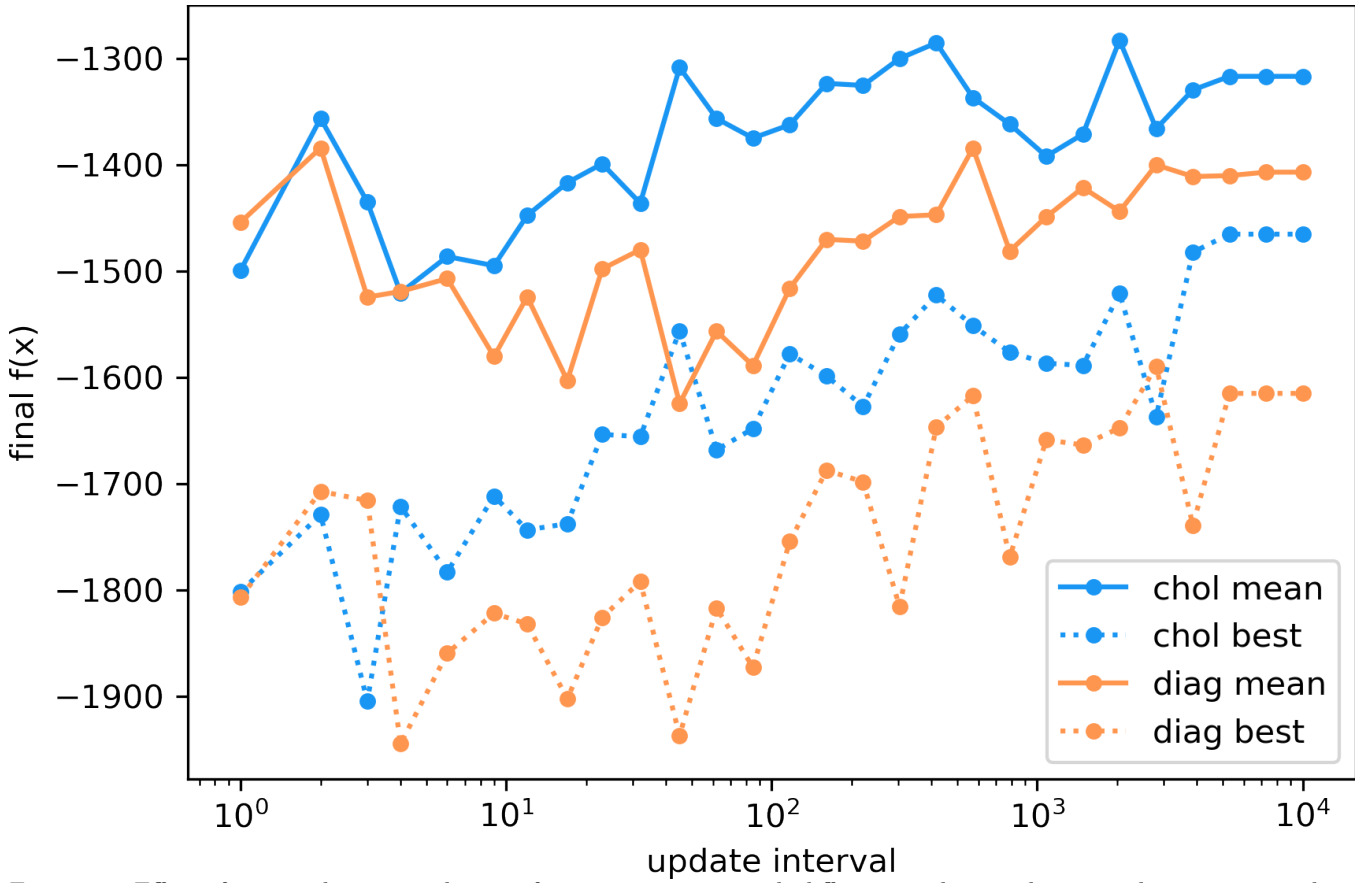

Figure 10: Effect of step-update-interval on performance. 30 runs with different random seeds are used to compute values at each point. The update interval of 10,000 is equivalent to the original step-update rule (only updating after accepted perturbations), and has clearly lower performance - showing the benefit of the `diversified-step-update-rule`

3.3.3 Temperature Schedule Hyperparameters: Markov Chain length and Alpha Parameter Optimisation

---

Maximum markov chain length and alpha both control the temperature schedule over the course of the Simulated Annealing algorithm. Maximum markov chain length controls the number of temperature steps, while alpha controls the size of the steps. If the markov chain length is large, then there are less total steps during the run, and thus the total temperature is reduced less. If alpha is small, the size of each temperature reduction step is large, and the temperature is reduced more over the run. This effect is shown in Figure 11. Having a well calibrated temperature schedule is important for the optimisation, as temperature controls the exploration exploitation trade-off of the program. Early in the program the temperature has to be set sufficiently high in order for the space to be explored, while later in the program the temperature has to be sufficiently low for the "best zone" to be narrowed in on. If the temperature is reduced too slowly or quickly, then the program will not sufficiently explore or exploit.

An example of the effects of maximum markov chain length and alpha on performance for `Diagonal-step` method and the `diversified-step-update-rule` are shown in Figure 12. Because both of these parameters affect the temperature schedule, there is rough equivalence in performance between certain schedules, for example (high MC length, low alpha) and (low MC length, high alpha) correspond to similar explore/exploit levels - this can be seen by the downwards sloping bands of colour in the contour plot. The average runtime is lowest for low values of both alpha and maximum markov chain length, as these lead to the system temperature becoming low quickly, resulting in convergence in lower number of steps.

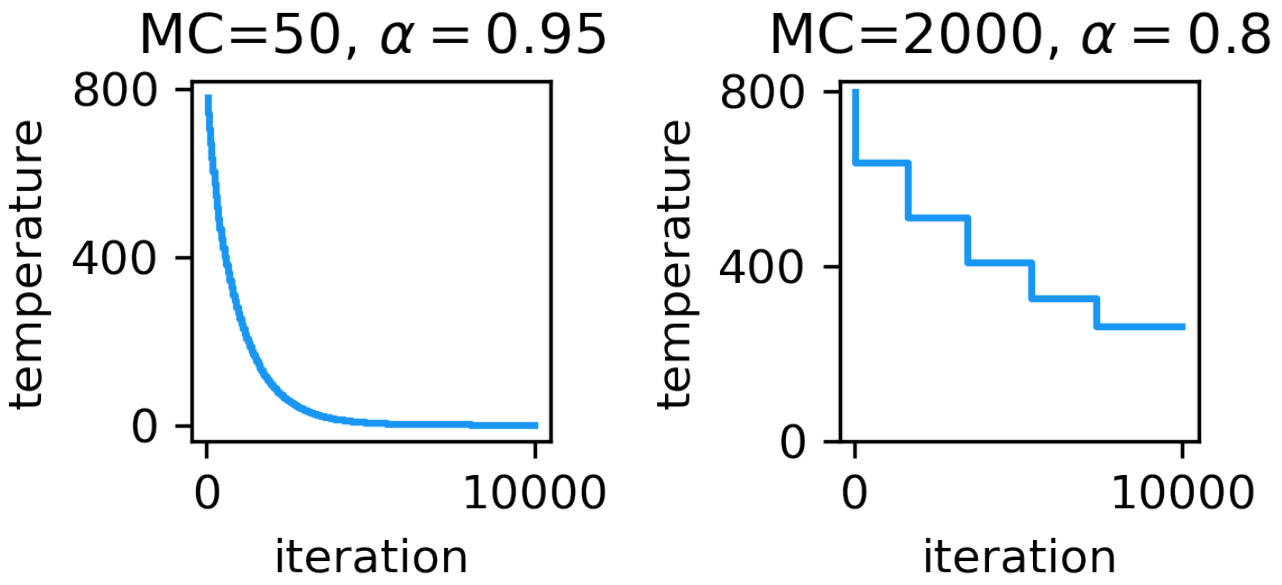

Figure 11: Effect of alpha and markov chain length (MC) on annealing schedule

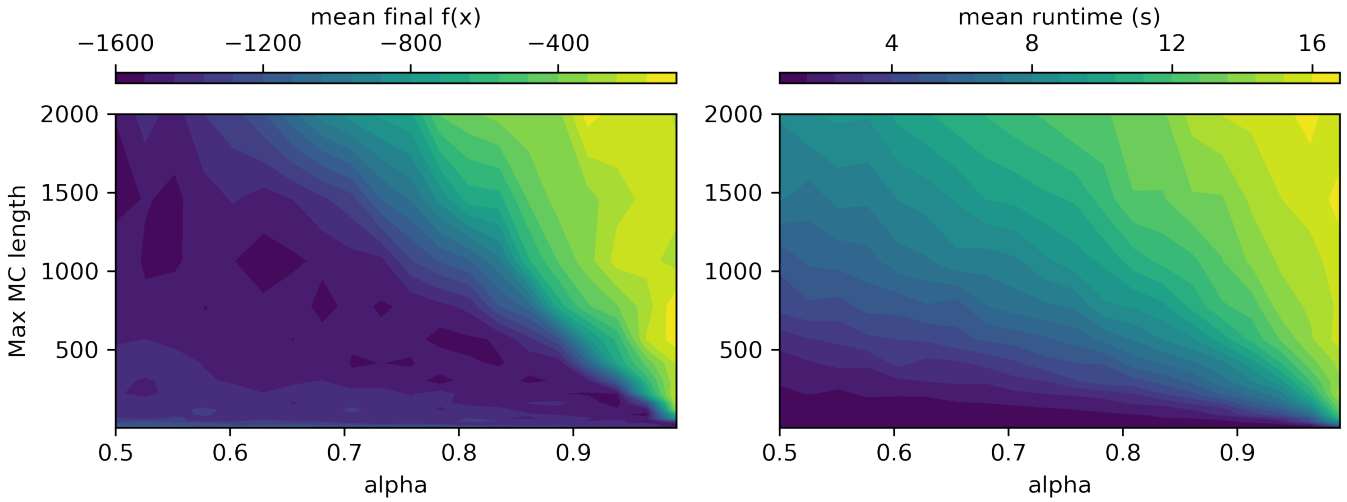

Figure 12: Contour plot for grid search of maximum markov chain length, and alpha value, for `Diagonal-step` method and `diversified-step-update-rule` (interval of 20). 30 runs with 30 different random seeds are used of estimation of mean objective value and mean time

3.3.4 Results with tuned hyperparameters

A grid search was performed for each of the step-methods, varying:

1. Temperature schedule hyperparameters: Markov chain length (30 values, logarithmically spaced between 5 and 2000), alpha (30 values, linearly space between 0.5 and 0.99)

2. `diversified-step-update-rule`: Interval (30 values, linearly spaced between 1 and 60) as well as testing without the rule (i.e. using the original rule of only updating the step size after perturbation acceptances).

The results for each method, with the optimised hyperparameters, are shown in Table 1.
The `diversified-step-update-rule` has clear benefits, significantly improving both the `Cholesky-step` and `Diagonal-step` method's mean objective function by over 1000. Both the best performing Simulated Annealing programs using the `Cholesky-step` and `Diagonal-step` methods have a lower mean objective function than the `Simple-step` method, showing the utility of the adaptive step size matrix. The `Diagonal-step` method had the best performance, both in terms of final value of the function, and in terms of runtime. Because *Rana's function* encourages exploration, the comparative advantage of the `Cholesky-step` method to more richly match the local topology (which is useful for exploitation) did not translate into better performance compared to the `Diagonal-step` method. Furthermore, the `Diagonal-step` method included useful adjustments of the acceptance probability, incorporating the step size to encourage exploration. The `Diagonal-step` method outperforms the `Cholesky-step` in terms of runtime because it has less computationally expensive operations (i.e doesn't have to calculate Cholesky decomposition). Although the `Diagonal-step` method requires more computation per step of the algorithm, it was also able to outperform the `Simple-step` method, by converging faster, and therefore requiring less steps.

Table 1: Results for different Simulated Annealing performance for different step-methods, with and without the diversified-step-update-rule, using tuned hyperparemeters (maximum markov chain length, alpha, and diversified-step-update-rule interval (when used). Performance metrics calculated over 30 runs with 30 different random seeds

step-method	**Cholesky**		**Diagonal**		**Simple**
diversified-step-update-rule interval	-	14	-	20	-
max markov chain length	776	776	1064	9	776
annealing alpha	0.77	0.5	0.55	0.99	0.61
mean final performance	-1409.06	-1578.82	-1551.19	-1666.23	-1481.48
std dev final performance	116.75	156.9	144.45	129.67	157.47
average runtime (s)	16.13	7.98	11.4	7.23	12.13

4 Evolutionary strategies

4.1 Implementation details

For each component of the Evolutionary Strategy Algorithm, the following methods were implemented, noting any deviations from the lecture slides by inserting a "⋆" at the start of where they are mentioned.

Mutation methods:

1. ⋆ `simple mutation`: Spherical Gaussian noise (non-adaptive)
2. `full covariance mutation`: Gaussian noise sampled from a covariance matrix formed with standard deviation and rotation angle strategy parameters. To ensure positive-definiteness, a small diagonal matrix is added repeatedly until the covariance is positive definite. ⋆ Off-diagonal elements of the covariance matrix are clipped to have their absolute values bounded by the minimum of the diagonal elements corresponding to the row and column number (e.g. element in row 2, column 3 is clipped to have it's absolute value bounded by the minimum of diagonal element 2 and diagonal element 3). This is a condition that covariance matrices hold, and helps encourage positive definiteness in a less computationally expensive manner than the original method of positive definiteness enforcement - which sometimes require repeated operations (the original method is still used in conjunction with the clipping). ⋆ Standard deviations clipped to be above 10^{-6} and below 1 to prevent mutation becoming too large or too small.
3. ⋆ `diagonal covariance mutation`: Same as `full covariance mutation` but with standard deviation strategy parameters only (i.e with rotation angles fixed at 0). Standard deviations clipped to be above 10^{-6} and below 1 to prevent mutation becoming too large or too small.

Recombination methods: Global discrete recombination for control parameters. Global intermediate recombination on strategy parameters.
Selection methods: μ, λ and $\mu + \lambda$
Convergence criteria: Absolute difference in objective function of parents $< 10^{-6}$. Maximum 10,000 iterations.
Bound enforcement: Repeated sampling of mutation until offspring within bounds.

4.2 Model Baseline on 2D Rana function

To show that the Evolution Strategy Algorithm is working as desired, an initial exploratory analysis is performed using `simple mutation` and μ, λ selection[5], on the 2D Rana function, as it allows for visualisation of the search patterns. The key areas that are indicative of desired performance are (1) minimisation of the objective function (2) balance of explore-exploit tradeoff (3) visibility of the components of the Evolution Strategy Algorithm: Mutation, Selection and Recombination.

Figure 13 shows a sample run, in which the objective function is successfully minimised (final value -491). Mutation and crossing over are the sources of variation of objective values - and their cumulative effects can be seen by the varying mean objective values of the offspring across time. As only the best children are selected as parents, the mean objective of the parents (per generation) is significantly below the mean objective of the offspring. Figure 13 is therefore also indicative of the selection working as desired. Because `simple mutationn` is used, the parents don't reach the convergence criterion (as they have non-negligible variance) even though functionally the algorithm has converged by 20 generations (the other mutation methods did in practice converge on the 2D Rana problem).

The search pattern plot (Figure 14), shows emphasis on exploration early in the run, where the solutions are spread throughout the space. The effect of selection is clear, with solutions at higher areas of the terrain becoming less prevalent as the program continues, as they are removed from the gene pool. The search pattern also shows that later in the algorithm, there is a strong focus on exploitation of the best solutions (with only 2 and 1 local zones focused on by generation 7, and 10 respectively). The symmetry of generation 4 along each axis is most likely a result of cross-over (e.g. as it allows for values of each element of various solutions to "swap", causing symmetry). Lastly, the effects of mutation are clear in the search patterns, where there are many values close (but distinctly separate) bunched at local minima.

[5]Configuration: 10 parents, 70 offspring, standard devation of each x element = 1% of the range of x

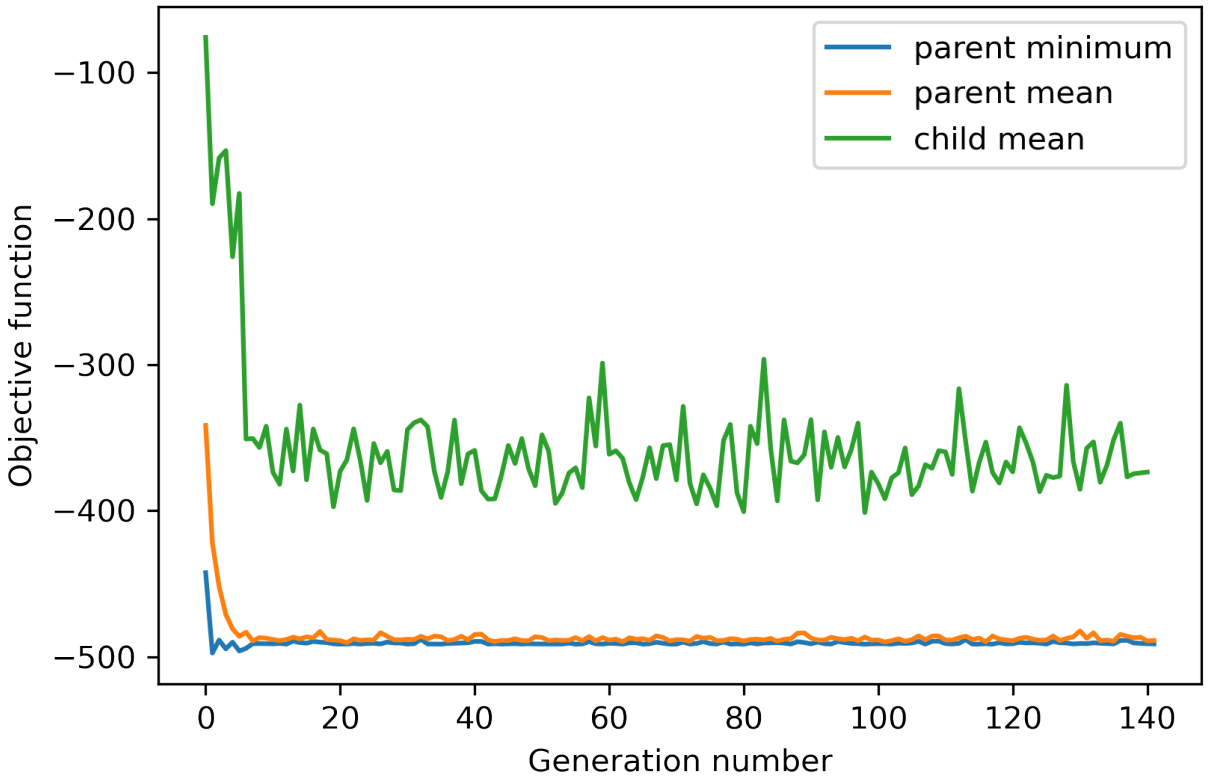

Figure 13: Evolution strategies method example performance on 2D rana function. `simple mutation`, μ, λ selection

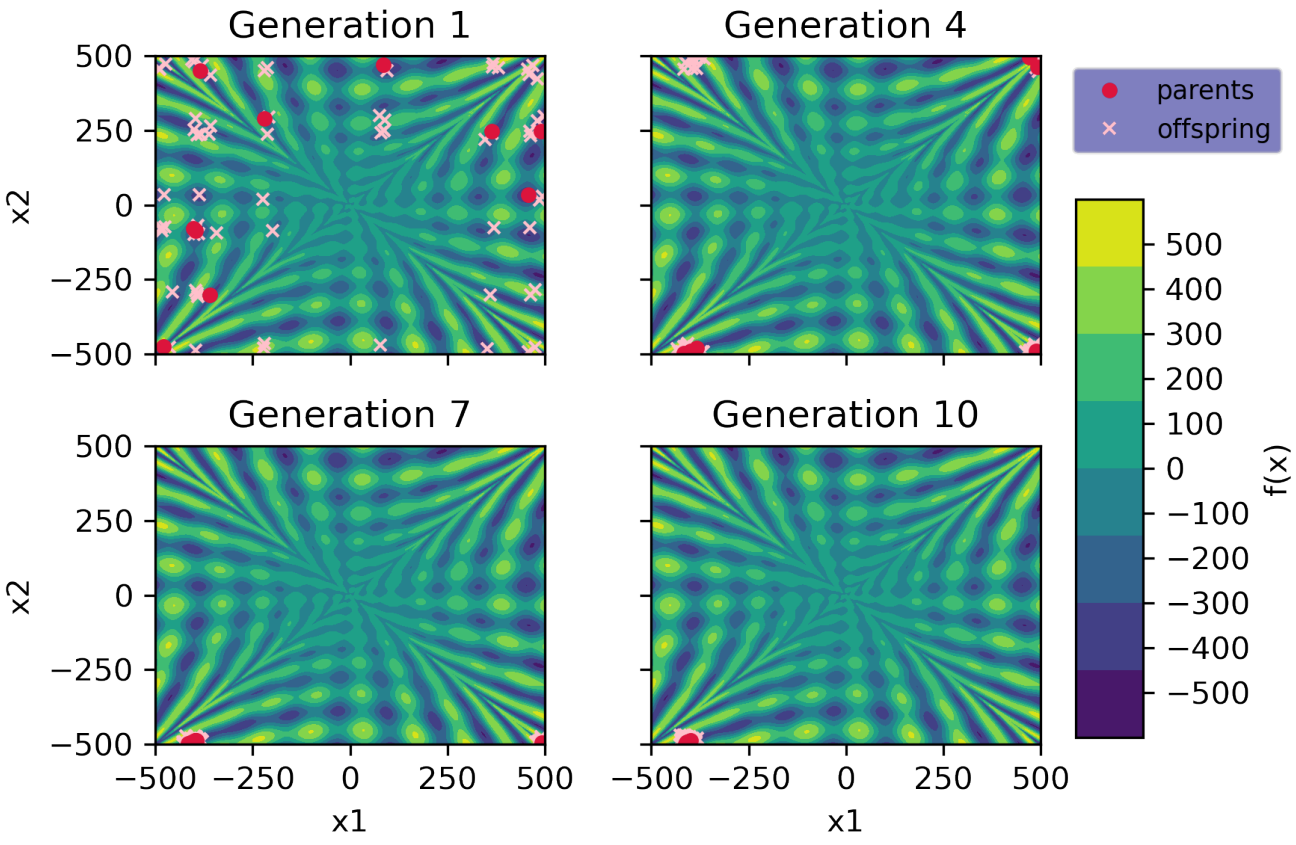

Figure 14: Evolution strategies method example search pattern on 2D rana function. `simple mutation`, μ, λ selection

4.3 Analysis on 5D Rana function

In this section, an initial presentation of the various methods for selection (μ, λ-selection and $\mu + \lambda$-selection), and mutations (`simple mutation`, `diagonal covariance mutation`, `full covariance mutation`) is given, with an emphasis on explaining their effects (performance comparisons are rather done after hyperparameter optimisation). The effects of the hyperparameters controlling the population dynamics are explored, and optimised (by grid search) for each combination of the selection and mutation methods. Finally, a comparison of the methods with the optimal hyperparameters is given.

4.3.1 Mutation method

In evolution by natural selection, dynamic or stressfull environments favour high levels of mutation, as useful adaptions are important for survival. On the other hand, stationary environments favour lower levels of mutation, as the parents typically are well suited to the environment, so children similar to the parents tend to have the highest fitness (Coyne 2010). This has a direct parallel to the Evolutionary Strategy algorithm, where the current values of the control parameters determines the "environment" (i.e. environment represented by location on control parameter landscape). Initially the control variables are in random locations that are relatively high in the landscape. As the run continues the algorithm will move significantly lower in the landscape, removing offspring similar to the original parents from the gene pool in the process. Thus the "environment" is initially dynamic, favouring high levels of mutation. Towards the end of the program, the parents are typically located in a good local minima, and change between generations becomes very small (as it is hard to find a lower point in the space). Thus later in the optimisation, the "environment" is stable, and lower levels of mutation are preferred.
The adaptive covariance methods are therefore useful as they allow the amount of mutation to adapt to the stage of the optimisation program. Furthermore, the adaptability of the magnitude mutation not only in total, but also with respect to specific elements of the control parameters (or with respect to linear combinations of control parameters in the case of `full covariance mutation`), allowing for higher levels of mutation along "directions" that are more promising.

The determinant of the covariance matrix is indicative of the total "amount" of mutation. Figure 16 shows this link, where smaller mutation-covariance determinant visibly leads to a smaller amount of variation in the offspring objective function[6]. Figure 16 shows that for both adaptive methods, the amount of mutation decreases significantly throughout the course of an example run - and thus, according to the description above, better suiting the later stationary state of the "environment".

[6] noting that selective pressure also has an effect here in reducing the variance. Comparing to Figure 13 shows that the **very** low variance comes from the lower levels of mutation (as if there is significant mutation, the standard deviation of the offspring never becomes especially small

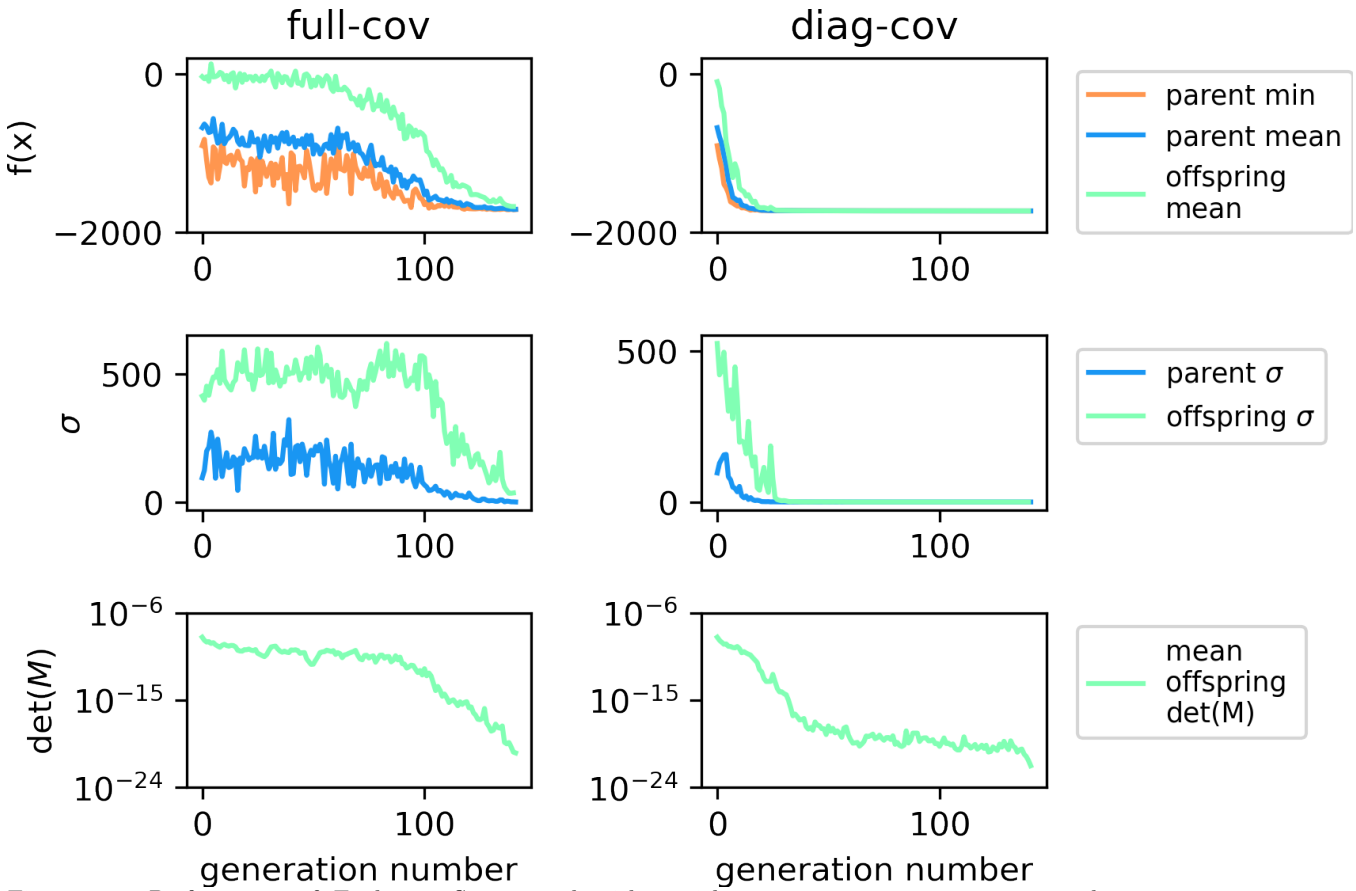

Figure 15: Performance of Evolution Strategy algorithm with `full covariance mutation` and `diagonal covariance mutation` for an example run. Note that M is the step size matrix.

4.3.2 Selection Method

μ, λ-selection is more reflective of evolution by natural selection (in which the parents of each generation die) and has the benefit of being better suited to "dynamic"[7] environments (there is a greater amount of accumulated mutation, giving a greater probability of useful new adaption). $\mu + \lambda$-selection has the advantage of never losing the "best" current solution, guaranteeing that the objective function will stay the same or decrease each generation - however it results in less exploration taking place, as some of the new parents in each generation are merely clones of the previous parent generation. In Figure 16, which shows an example run of both selection methods, the minimum parent of the μ, λ-selection often increases, where with μ, λ-selection the minimum parent objective is always lower or the same as the previous generation. In the given example run (not necessarily in general), the μ, λ-selection has a better final objective value, due to its superior exploration of the space.

[7] dynamic evolutionary environments defined in Section 4.3.1

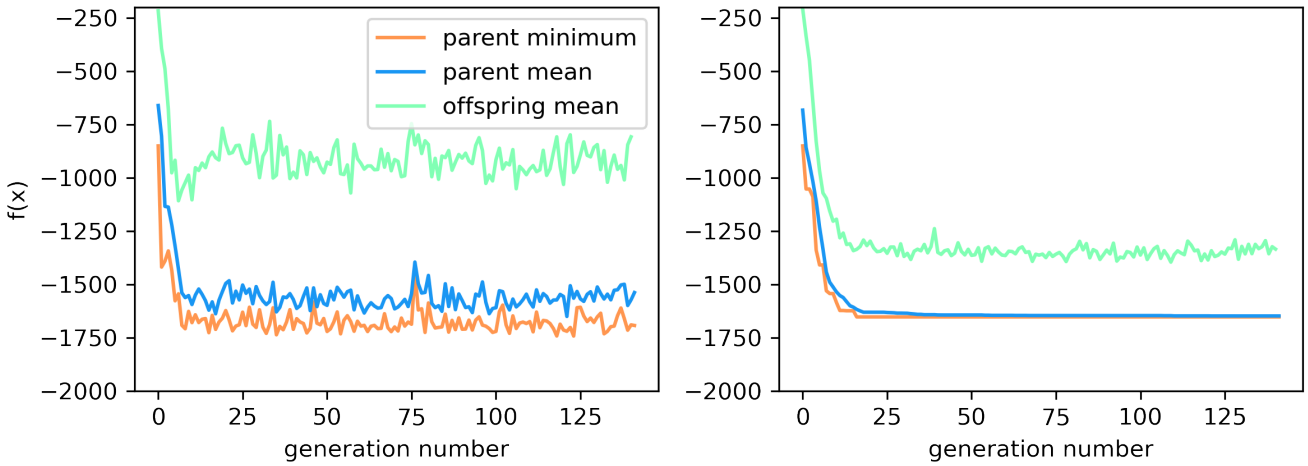

Figure 16: Performance of Evolution Strategy algorithm with μ, λ-selection and $\mu + \lambda$-selection

4.3.3 Population dynamics

The child to parent ratio controls the selective pressure on the system. If the child to parent ratio is high, because only a small fraction of children survive each generation, the selective pressure is high (only the very fittest survive), while if the child to parent ratio is relatively low, then many offspring survive and the selective pressure is relatively low. This relates the exploration exploitation trade-off - higher selective pressure causes a larger focus on narrow searches in the current best local optima (**exploitation**), while lower selective pressure allows for a greater **exploration** of the space. At the extreme of a 1-1 parent to child ratio, the algorithm becomes a random walk ("pure" exploration).

The effect of child to parent ratio was explored by holding the number of offspring at 100, and changing the number of parents per generation - as this allows for the number of generations (over 10,000 iterations[8]), and the amount of mutation per generation to be held constant.

The effect of child to parent ratio is demonstrated in Figure 17. The fixed-diagonal Gaussian mutation was used for this figure - as this prevents interaction effects between the mutation method and the offspring-to-parent-ratio, so allows for a simpler isolation of the effects of offspring-to-parent-ratio. In the left hand side plot, the child-to-parent-ratio is relatively low (2:1) and the following trends are clear (1) the decrease in the parents' average objective function over generations is relatively slow - from lower selective pressure creating less downwards force on the parents objectives (some parents with only mildly low objective functions are accepted each iteration). (2) The variance of the parent population is relatively high - because there are more parents, it is more likely that a parent near a different local minima to the current best will be selected. Conversely, in the right hand plot the child-to-parent-ratio is high (20:1) (1) the parents' average objective function decreases very rapidly (and then plateaus) (only the lowest objective values are selected, so the objective function quickly decreases). (2) the variance in the parents' objective functions quickly becomes very low, as the high selective pressure causes the parents to bunch around a single local optimal. Overall it is clear that the low child-to-parent ratio focuses more on exploration, and the high child to parent ratio on exploitation.

Holding the parent to child ratio and the maximum number of function evaluations constant, the population size (both parents and offspring) controls the exploration-exploitation in a different way. If the number of offspring is large then there is a lower number of total generations[9] (as the 1000 function evaluations are "consumed" at a higher rate per generation) and if the number of offspring is small then there are many generations. As early generations focus more of exploration, and later generations on explotation, if most of the children are contained in early generations (if there are many offspring), there is a greater focus on exploration (and correspondingly, less children $\implies$ more exploitation). Another way this has an effect is through the number of parents - if there are many parents, then a higher number of promising local zones are passed from generation to generation. Thus for a large space with many local minima, a larger population is preferred.

[8] if convergence is reached early, then there will be less generations

[9] assuming that 10,000 function evaluations are performed (if convergence occurs early then there are less function evaluations, and less generations)

Figure 18 shows an example of the joint effects of parent to child ratio and offspring number on performance for the Evolution Strategy algorithm with `diagonal covariance` mutation and μ, λ-selection, representing a subset (`diagonal covariance` mutation and μ, λ-selection) of the grid search optimisation performed on the population dynamics hyperparameters. In practice these plots and the resultant best population dynamics hyperparameters differed significantly for different methods. This plot does show some the general trends mentioned above for the population dynamics parameters, namely (1) decreased performance if the number of children becomes too large corresponding to a lack of exploitation (2) decreased performance if the number of children is too small corresponding to a lack of exploration and (3) decreased performance if the child to parent ratio is too small, corresponding to not enough selective pressure.

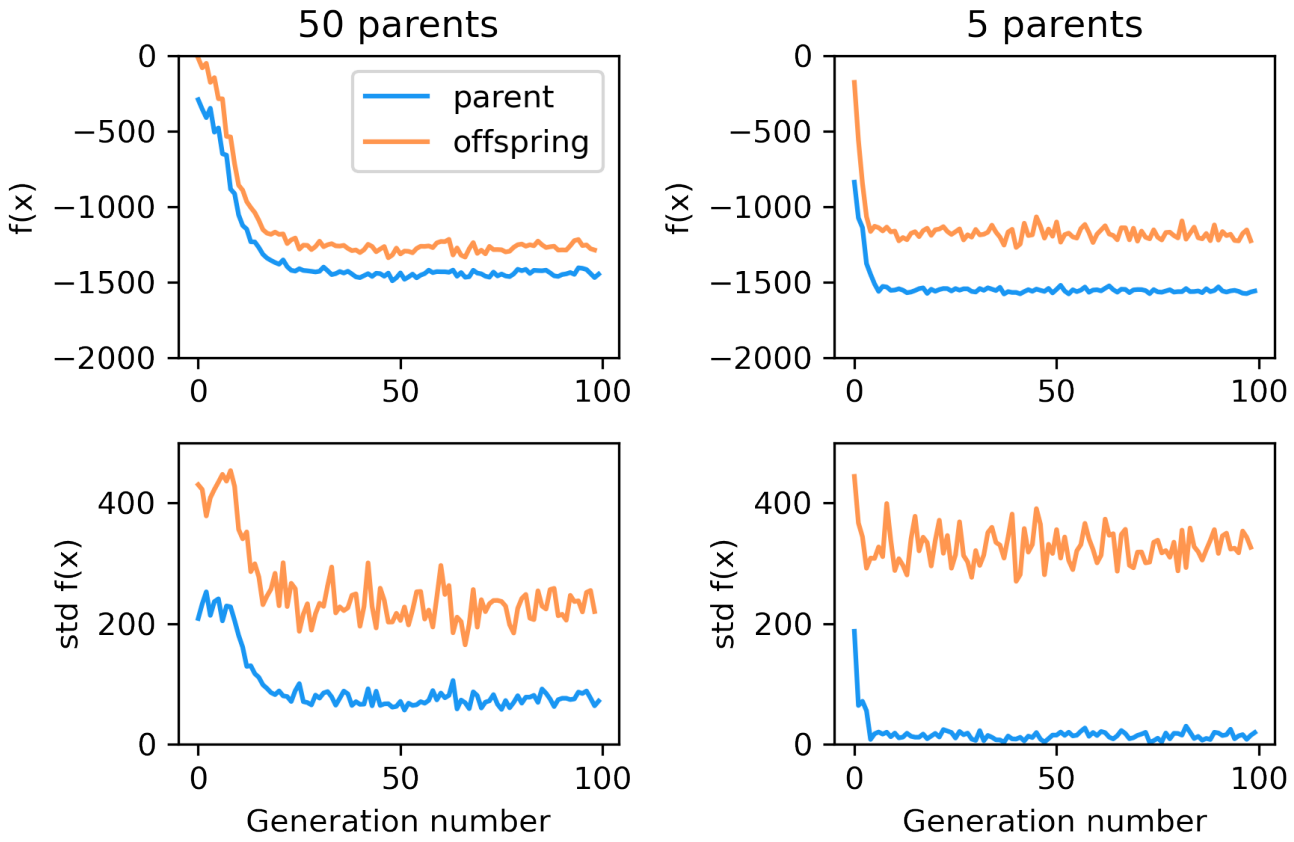

Figure 17: Illustration of the effect of child to parent ratio on performance. Number of offspring kept constant (100 offspring) Using non-adaptive mutation method, μ, λ-selection.

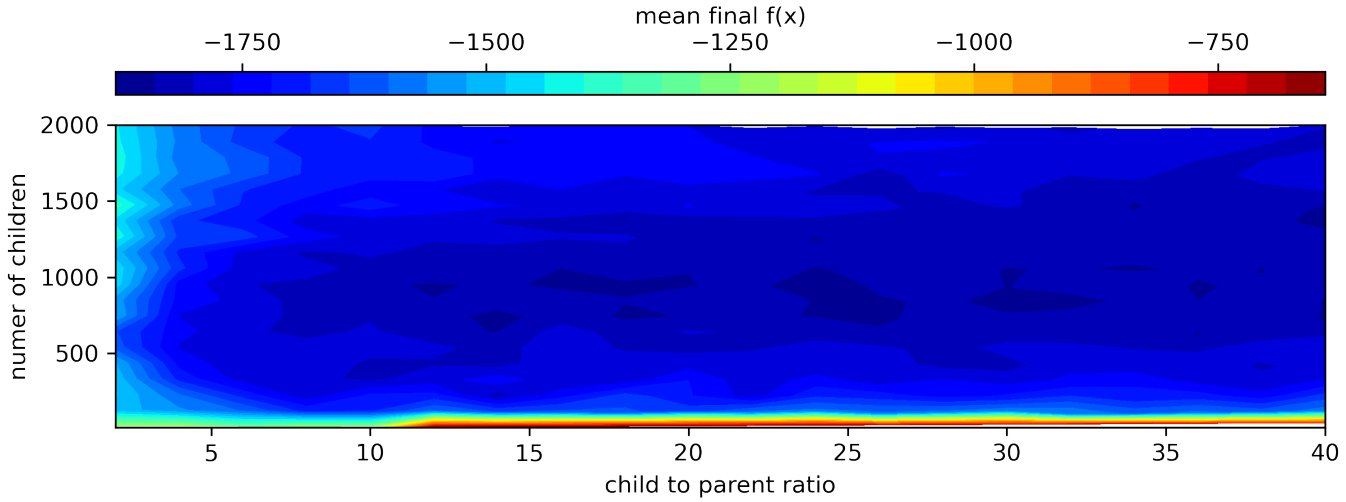

Figure 18: Contour plot of performance of the Evolution Strategy algorithm with `diagonal covariance` mutation and μ, λ-selection. Values calculated through averaging the results of 30 runs with 30 different random seeds.

4.3.4 Final Comparison with optimised population dynamics hyperparameters

A grid search optimisation was performed on the population dynamics hyperparameters for each of the mutation methods, and selection methods. The search grid was composed of 30 equally spaced values for offspring number (between 20 and 2000), and offspring per parent (between 2 and 40), with performance aggregated over 30 runs per point.

Table 2 below shows the performance results with the optimised hyperparameter settings. The `diagonal covariance mutation` method and $\mu + \lambda$ selection each generally had the best performance (with their combination yielding the best results). The `diagonal covariance mutation` method's high performance is most likely because (1) it has the benefit of an adaptive mutation, allowing it to adjust the level of mutation to fit the "evolutionary environment" (2) it has a lower number of parameters than the `full covariance mutation` and therefore is more robust. The `full covariance mutation` also has a significantly higher average runtime, due to the more expensive operation of required to (1) calculate the covariance matrix from the rotation and standard deviation strategy parameters and (2) enforce positive definiteness. $\mu + \lambda$ selection, was preferred because of the advantage of never "losing" the best solutions throughout the optimisation, and through selecting a high number of parents, the downside of $\mu + \lambda$ selection (less exploration) is minimised.

Table 2: Final results (calculated over 30 runs with 30 different random seeds), for each combination of mutation and selection methods, using optimised population dynamics hyperparameters.

mutation method	complex	complex	diagonal	diagonal	simple	simple
selection method	μ, λ	$\mu + \lambda$	μ, λ	$\mu + \lambda$	μ, λ	$\mu + \lambda$
Offspring per parent	32	14	28	40	24	40
number of offspring	96	112	952	840	936	840
parent number	3	8	34	21	39	21
mean f(x)	-1695.8	-1774	-1898.09	-1905.97	-1864.4	-1878.6
std dev f(x)	134.48	63.18	18.05	23.64	26.05	30.77
average runtime (s)	7.34	9.78	1.08	1.04	1.06	1.15

5 Overall Comparison

A comparison of the best performing Simulated Annealing and Evolution Strategy algorithms on the 5D *Rana's function* is shown in Table 3. Both algorithms are able to significantly outperform random search (which can be thought of an algorithm that focuses purely on exploration). Although this is not particularly impressive, it does provide a useful "sanity check" baseline, indicating that the algorithms are completing some functionality beyond randomly searching through the space. The Evolution Strategy algorithm is far superior to Simulated Annealing, both in terms of minimising the objective function and in terms of average runtime. The lower runtime of the Evolution Strategy Algorithm is due to it's ability to run function evaluations of each generation in parallel.

Rana's function has a roughly linear relationship between the global local minima's objective value, and the number of dimensions (Vanaret et al. 2020). Figure 19 shows the performance of the Evolution Strategy algorithm is consistently superior to random search and Simulated Annealing for an increasing number of dimensions of the *Rana's function*, with the performance difference increasing with the number of dimensions, with random search and Simulated Annealing's performance becoming quickly sub-linear. This shows that the more "difficult" the problem, the better the Evolution Strategy algorithms relative performance.

Table 3: Simulated Annealing and Evolution Strategy performance with best performing configurations, benchmarked against a random search (uniform sampling within the space for 10,000 iterations)

Method	**Simulated Annealing**	**Evolution Strategy**	**Random Search**
mean f(x)	-1666.23	-1905.97	-1498.15
std dev f(x)	129.67	23.64	83.08
average runtime (s)	7.23	1.04	0.27

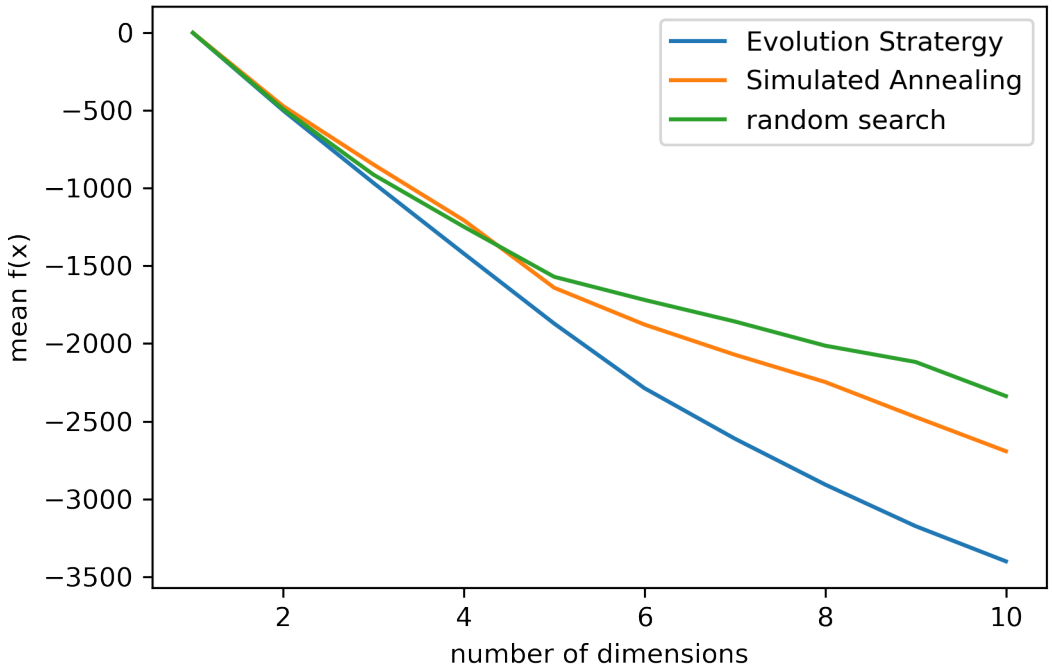

Figure 19: Performance of Simulated Annealing, Evolution Strategy and Random Search optimisation with increasing dimension of *Rana's function*. Performances calculated for 30 runs with 30 different random seeds

References

[1] Franco Busetti. "Simulated annealing overview". In: (2003). URL: https://www.researchgate.net/profile/Franco_Busetti/publication/238690391_Simulated_annealing_overview/links/5c0e6c72299bf139c74de536/Simulated-annealing-overview.pdf.

[2] Jerry A Coyne. *Why evolution is true*. Oxford University Press, 2010.

[3] Scott Kirkpatrick. "Optimization by simulated annealing: Quantitative studies". In: *Journal of statistical physics* 34.5-6 (1984), pp. 975–986.

[4] Charlie Vanaret et al. "Certified global minima for a benchmark of difficult optimization problems". In: *arXiv preprint arXiv:2003.09867* (2020).

[5] Darrell Whitley et al. "Evaluating evolutionary algorithms". In: *Artificial Intelligence* 85.1 (1996), pp. 245–276. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(95)00124-7. URL: http://www.sciencedirect.com/science/article/pii/0004370295001247.

6 Appendix

The code has been written with descriptive variable and function names, with the aim of making the steps clear without the need for excessive commenting (although comments are still given as an additional aid).

6.1 Simulated Annealing code

```
import numpy as np
from collections import deque

class SimulatedAnnealing:
    """
    x is transformed from the original bounds, to be bounded by (-1, 1)
    Before outputs are given x is re-transformed back into the original form

    This class includes methods (using restarts, adaptive cooling) that aren't explored
    in the report for the sake of brevity
    """
    def __init__(self, x_length, x_bounds, objective_function,
                 pertubation_method="simple",
                 annealing_schedule="simple_exponential_cooling", with_restarts=False,
                 archive_minimum_acceptable_dissimilarity=0.2,
                 maximum_markov_chain_length=10, bound_enforcing_method="clipping",
                 maximum_archive_length=20, step_size_initialisation_fraction_of_range=0.5,
                 annealing_alpha=0.95,
                 maximum_function_evaluations=10000,
                 cholesky_path_length=5,
                 pertubation_alpha = 0.1, pertubation_omega = 2.1,
                 convergence_min_improvement = 1e-6,
                 update_step_size_when_not_accepted_interval = 1,
                 ):

        self.x_length = x_length    # integer containing length/dimension of array x
        self.x_bounds = x_bounds    # tuple containing the original bounds to x
        self.x_range_internal = 2    # after being transformed to be between -1 and 1
        self.objective_function_raw = objective_function
        self.pertubation_method = pertubation_method
        self.annealing_schedule = annealing_schedule
        self.markov_chain_maximum_length = maximum_markov_chain_length
        self.acceptances_minimum_count = round(0.6 * maximum_markov_chain_length)   # 0.6 is a
        heuristic from lectures
        self.convergence_min_improvement = convergence_min_improvement
        self.convergence_evaluation_window = int(maximum_function_evaluations/10)
        # bound enforcement allows for clipping to be used (speeding up the clipping, but
        introducing bias)
        # the bound_enforcing_method is set to "not_clipping" throughout the report
        self.bound_enforcing_method = bound_enforcing_method
```

```
        # the below variable controls the "diversified-step-update-rule"
        # i.e. how often the step-size control matrix is updated per number of iterations
        # a value of 1 updates every step,
        # a value of False only updates on acceptances (turns off the "step-update-rule")
        self.update_step_size_when_not_accepted_interval =
        ↪  update_step_size_when_not_accepted_interval

        # initialise archive and parameters determining how archive is managed
        self.archive = []   # list of (x, objective value) tuples
        self.archive_maximum_length = maximum_archive_length
        self.archive_minimum_acceptable_dissimilarity =
        ↪  archive_minimum_acceptable_dissimilarity
        self.archive_similar_dissimilarity = archive_minimum_acceptable_dissimilarity

        # initialise histories and counters
        # these are useful for inspecting the performance of the algorithm after a run
        # and are used within the program (e.g. markov chain length)
        self.accepted_objective_history = []
        self.acceptences_locations = []
        self.accepted_x_history = []
        self.objective_history = []
        self.x_history = []
        self.alpha_history = []
        self.temperature_history = []
        self.step_size_matrix_history = []
        self.step_size_update_locations = []
        self.probability_of_acceptance_history = []
        self.iterations_without_acceptance = 0
        self.Markov_chain_length_current = 0    # initialise to 0
        self.acceptances_count_current_chain = 0  # initialise to 0
        self.acceptances_total_count = 0    # initialise to 0
        self.objective_function_evaluation_count = 0    # initialise to 0

        # option to restart the algorithm if no progress is made
        self.restart_count = 0
        self.max_iterations_without_acceptance_till_restart =
        ↪  int(maximum_function_evaluations/10)
        self.with_restarts = with_restarts

        if pertubation_method == "simple":  # step size just a constant in this case
            self.step_size_matrix = 2*step_size_initialisation_fraction_of_range
            # stays at initialisation in simple pertubation method
        elif pertubation_method == "Cholesky" or "Diagonal":
            self.pertubation_alpha = pertubation_alpha
            self.pertubation_omega = pertubation_omega
            self.recent_x_history = deque(maxlen=cholesky_path_length)  # used for local
            ↪  topology covariance
            if pertubation_method == "Cholesky":
                self.step_size_control_matrix = (np.eye(x_length) *
```

```
                                            step_size_initialisation_fraction_of_range *
                                            ↪ self.x_range_internal) ** 2
            self.step_size_matrix = np.linalg.cholesky(self.step_size_control_matrix)
        else:  # Diagonal
            self.step_size_matrix = np.eye(x_length) * \
                                    step_size_initialisation_fraction_of_range *
                                    ↪ self.x_range_internal


    if annealing_schedule == "simple_exponential_cooling":
        self.annealing_alpha = annealing_alpha   # alpha is a constant in this case
    elif annealing_schedule == "adaptive_cooling":
        self.current_temperature_accepted_objective_values = []   # alpha calculated off
        ↪ standard deviation of this

    self.objective_function_evaluation_max_count = maximum_function_evaluations


def objective_function(self, x):
    """
    Wrapper for the objective function calls, adding some extra functionality
    """
    self.objective_function_evaluation_count += 1   # increment by one everytime objective
    ↪ function is called
    x_interp = np.interp(x, [-1, 1], self.x_bounds) # interpolation done here to pass the
    ↪ objective function x correctly interpolated
    result = self.objective_function_raw(x_interp)
    return result


def run(self):
    """
    This function run's the major steps of the Simulated Annealing algorithm
    # major steps in the algorithm are surrounded with a #******************************
    # other parts of this function are merely storing histories, and updating counters
    """
    # initialise x and temperature
    self.x_current = self.initialise_x()
    self.objective_current = self.objective_function(self.x_current)
    self.initialise_temperature(self.x_current, self.objective_current)
    self.archive.append((self.x_current, self.objective_current))  # initialise archive
    if self.pertubation_method == "Cholesky":
        self.recent_x_history.append(self.x_current)
    done = False    # initialise, done = True when the optimisation has completed
    while done is False:
        # **********************   PERTUBATION      **********************
        x_new = self.perturb_x(self.x_current)
        # ****************************************************************
        objective_new = self.objective_function(x_new)
```

```
            delta_objective = objective_new - self.objective_current
            delta_x = x_new - self.x_current
            self.objective_history.append(objective_new)
            self.x_history.append(x_new)

            # *****************    Asses Solution    *************************
            # accept change if there is an improvement, or probabilisticly (based on given
             ↪  temperature)
            if self.accept_x_update(delta_objective, delta_x):
            # ********************************************************************
                self.update_archive(x_new, objective_new)
                if self.pertubation_method == "Cholesky":
                    # store recent x values used to calculate covariance matrix
                    self.recent_x_history.append(x_new)
                elif self.pertubation_method == "Diagonal":
                    self.latest_accepted_step = x_new - self.x_current

                self.x_current = x_new   # update x_current
                self.objective_current = objective_new
                if self.annealing_schedule == "adaptive_cooling":
                    # record accepted objective values in the current chain
                    self.current_temperature_accepted_objective_values.append(
                        self.objective_current)
                self.accepted_objective_history.append([objective_new,
                 ↪  self.objective_function_evaluation_count])
                self.acceptences_locations.append(self.objective_function_evaluation_count)
                self.accepted_x_history.append(x_new)
                self.acceptances_count_current_chain += 1 # in current markov chain
                self.acceptances_total_count += 1
                self.iterations_without_acceptance = 0
                # **********************Update Step size ***************
                self.update_step_size()
                # ***************************************************
            else:
                self.iterations_without_acceptance += 1
                # **********************diversified-step-update ***************
                # update according to folding interval, using the diversified-step-update-rule
                if self.update_step_size_when_not_accepted_interval is not False and \
                        self.objective_function_evaluation_count %
                         ↪  self.update_step_size_when_not_accepted_interval == 0:
                    self.update_step_size()
                # ***************************************************
            self.Markov_chain_length_current += 1
            # ***************  Update Temperature  *****************
            # update temperature if need be
            # also checks for convergence
            done = self.temperature_scheduler()
            # ***************************************************
            if self.with_restarts:
```

```
                self.asses_restart()
            if self.restart_count > 5:
                print("max restarts reached, stopping early")
                break

        return np.interp(self.x_current, [-1, 1], self.x_bounds), self.objective_current

    def accept_x_update(self, delta_objective, delta_x):
        """
        returns True if the latest pertubation is accepted
        """
        if delta_objective < 0:
            return True
        else:
            if self.pertubation_method == "Diagonal":
                probability_of_accept = np.exp(-delta_objective /
                ↪  (self.temperature*np.sqrt(np.sum(delta_x**2))))
            else:
                probability_of_accept = np.exp(-delta_objective / self.temperature)
            self.probability_of_acceptance_history.append([np.clip(probability_of_accept, 0,
            ↪  1), self.objective_function_evaluation_count])
            if probability_of_accept > np.random.uniform(low=0, high=1):
                return True
            else:
                return False

    def initialise_x(self):
        # initialise x randomly within the given bounds
        return np.random.uniform(low=-1, high=1, size=self.x_length)

    def initialise_temperature(self, x_current, objective_current, n_steps=60,
    ↪  average_accept_probability=0.8):
        """
        Initialises system temperature using Kirkpatrick method
        As all x's are initially accepted, x does a random walk, so changes in x are discarded
        """
        objective_increase_history = []  # if many samples are taken then this could be changed
        ↪  to running average
        for step in range(1, n_steps+1):
            x_new = self.perturb_x(x_current)
            objective_new = self.objective_function(x_new)
            if objective_new > objective_current:
                objective_increase_history.append(objective_new - objective_current)
            if step == n_steps:
                self.latest_accepted_step = x_new - self.x_current
            x_current = x_new
            objective_current = objective_new

        initial_temperature = - np.mean(objective_increase_history) /
        ↪  np.log(average_accept_probability)
```

```
        self.temperature = initial_temperature
        self.temperature_history.append([self.temperature, self.acceptances_total_count,
        ↪  self.objective_function_evaluation_count])

    def is_positive_definate(self, matrix):
        try:
            np.linalg.cholesky(matrix)
            return True
        except:
            return False

    def update_step_size(self):
        """
        Update the matrix controlling the step size
        """
        # record when step size updates happened
        self.step_size_update_locations.append(self.objective_function_evaluation_count)
        if self.pertubation_method == "simple":
            return  # no update with the simple method
        elif self.pertubation_method == "Cholesky":
            covariance = np.cov(self.recent_x_history, rowvar=False)
            # prevent covariance from becoming too large by clipping
            covariance = np.clip(covariance, -self.x_range_internal / 2, self.x_range_internal
            ↪  / 2)
            self.step_size_control_matrix = (1 - self.pertubation_alpha) *
            ↪  self.step_size_control_matrix + \
                                        self.pertubation_alpha * self.pertubation_omega *
                                        ↪  covariance
            # conservative clipping preventing step size control matrix from exploding or
            ↪  getting too small
            self.step_size_control_matrix =
            ↪  self.step_size_control_matrix.clip(-self.x_range_internal * 2,
            ↪  self.x_range_internal * 2)
            i = 0
            while not self.is_positive_definate(self.step_size_control_matrix):
                i += 1
                self.step_size_control_matrix += np.eye(self.x_length)*1e-6*10**i  # to make
                ↪  positive definate
                if i > 7:
                    raise Exception("couldn't get positive definate step size control matrix")
            if np.linalg.det(self.step_size_matrix) < 1e-16:
                # increase step size if determinant falls below 1e-6
                self.step_size_control_matrix = self.step_size_control_matrix * 1.1
                # now have to enforce positive definateness again
                while not self.is_positive_definate(self.step_size_control_matrix):
                    i += 1
                    self.step_size_control_matrix += np.eye(self.x_length) * 1e-6 * 10 ** i
                    if i > 7:
                        raise Exception("couldn't get positive definate step size control
                        ↪  matrix")
```

```
            self.step_size_matrix = np.linalg.cholesky(self.step_size_control_matrix)
            self.step_size_matrix_history.append(self.step_size_matrix)

        elif self.pertubation_method == "Diagonal":

            self.step_size_matrix = \
                (1-self.pertubation_alpha)*self.step_size_matrix + \
                ↪  np.diag(self.pertubation_alpha*self.pertubation_omega*np.abs(self.latest_accepted_ste
            # conservative clipping to prevent step size becoming too small or large
            self.step_size_matrix = np.clip(self.step_size_matrix, self.x_range_internal *
            ↪  1e-16, self.x_range_internal * 2)

            if np.linalg.det(self.step_size_matrix) < 1e-16:
            #  increase step size if determinant falls below 1e-6
                self.step_size_matrix = self.step_size_matrix*1.1
            self.step_size_matrix_history.append(np.diag(self.step_size_matrix))

    def perturb_x(self, x):
        if self.pertubation_method == "simple":
            u_random_sample = np.random.uniform(low=-1, high=1, size=self.x_length)
            x_new = x + self.step_size_matrix * u_random_sample  # constant step size
            if self.bound_enforcing_method == "clipping":
                return np.clip(x_new, -1, 1)
            else:
                while max(x_new) > 1 or min(x_new) < -1:
                    indxs_breaking_bounds = np.where((x_new > 1) + (x_new < -1) == 1)
                    u_random_sample = np.random.uniform(low=-1, high=1,
                    ↪  size=indxs_breaking_bounds[0].size)
                    x_new[indxs_breaking_bounds] = x[indxs_breaking_bounds] +
                    ↪  self.step_size_matrix * u_random_sample


        elif self.pertubation_method == "Cholesky":
            u_random_sample = np.random.uniform(low=-np.sqrt(3), high=np.sqrt(3),
            ↪  size=self.x_length)
            x_new = x + self.step_size_matrix@u_random_sample
            if self.bound_enforcing_method == "clipping":
                return np.clip(x_new, -1, 1)
            else:
                if max(x_new) > 1 or min(x_new) < -1:
                     x_new = self.perturb_x(x)   # recursively call perturb until sampled
                     ↪  within bounds

        elif self.pertubation_method == "Diagonal":
            u_random_sample = np.random.uniform(low=-1, high=1, size=self.x_length)
            x_new = x+self.step_size_matrix@u_random_sample
            if self.bound_enforcing_method == "clipping":
                return np.clip(x_new, -1, 1)
```

```
            else:
                while max(x_new) > 1 or min(x_new) < -1: # only sample specific indices not within bounds
                    indxs_breaking_bounds = np.where((x_new > 1) + (x_new < -1) == 1)
                    u_random_sample = np.random.uniform(low=-1, high=1, size=indxs_breaking_bounds[0].size)
                    x_new[indxs_breaking_bounds] = \
                        x[indxs_breaking_bounds] + \
                        np.diag(np.diag(self.step_size_matrix)[indxs_breaking_bounds])\
                        @u_random_sample
        return x_new


    def asses_restart(self, min_difference = 0.01):
        length = self.markov_chain_maximum_length  # base on length of markov chain
        if len(self.objective_history) % length == 0 and \
            len(self.accepted_objective_history) > self.markov_chain_maximum_length:
            if max(self.accepted_objective_history_array[-length:, 0]) - min(self.accepted_objective_history_array[-length:, 0]) < min_difference:
                # then rebase from best archive solution
                x_restart = self.archive_x[np.argmax(self.archive_f), :]
                print("restarted due to minimal progress")
                self.restart_count += 1
                self.x_current = x_restart
            elif self.iterations_without_acceptance > self.max_iterations_without_acceptance_till_restart:
                x_restart = self.archive_x[np.argmax(self.archive_f), :]
                print(f"restarted due to {self.max_iterations_without_acceptance_till_restart} iterations "
                      f"without acceptence")
                self.restart_count += 1
                self.x_current = x_restart


    def temperature_scheduler(self):
        if self.Markov_chain_length_current > self.markov_chain_maximum_length or \
                self.acceptances_count_current_chain > self.acceptances_minimum_count:
            if self.annealing_schedule == "simple_exponential_cooling":
                self.temperature = self.temperature * self.annealing_alpha
            elif self.annealing_schedule == "adaptive_cooling":
                if len(self.current_temperature_accepted_objective_values) <= 1:
                    # if no values have been accepted, then don't change alpha
                    # algorithm most likely close to convergence
                    self.alpha = 1
                    self.alpha_history.append([self.alpha, self.objective_function_evaluation_count])
                else:
                    latest_temperature_standard_dev = np.std(self.current_temperature_accepted_objective_values)
```

```
                    # use adaptive temperature rule
                    self.alpha = np.max([0.5,
                    ↪  np.exp(-0.7*self.temperature/latest_temperature_standard_dev)])
                    self.alpha_history.append([self.alpha,
                    ↪  self.objective_function_evaluation_count])
                self.temperature = self.temperature * self.alpha
            if np.isnan(self.temperature):
                self.temperature = 1e-16
                print("minimum temp reached")
            self.current_temperature_accepted_objective_values = []     # reset
            self.temperature_history.append([self.temperature, self.acceptances_total_count,
            ↪  self.objective_function_evaluation_count])
            self.Markov_chain_length_current = 0    # restart counter
            self.acceptances_count_current_chain = 0  # restart counter
        done = self.get_halt()
        if done is True:
            # add final values to make plotting histories easier
            self.temperature_history.append(
                [self.temperature, self.acceptances_total_count,
                ↪  self.objective_function_evaluation_count])
            self.accepted_objective_history.append([self.objective_current,
            ↪  self.objective_function_evaluation_count])
            self.acceptences_locations.append(self.objective_function_evaluation_count)
            self.accepted_x_history.append(self.x_current)
        return done

    def get_halt(self):
        """
        1.  first check convergence, converge if over the last evaluation window (5% of total
↪  max function evals), the
         diffrence between the maximum and minimum accepted values (within the window) is below
↪  the threshold defined
         by self.convergence_min_improvement (typically set to 1e-8)
        2. If the maximum number of function evaluations has been reached (typically set to 10
↪  000) then end program
        """
        if self.objective_function_evaluation_count % self.convergence_evaluation_window == 0:
            # only make this check every self.convergence_evaluation_window number of
            ↪  iterations
            acceptences_locations_array = np.array(self.acceptences_locations)
            acceptence_indx_over_window = \
            np.arange(len(acceptences_locations_array))[
                        acceptences_locations_array >
                        self.objective_function_evaluation_count -
                        ↪  self.convergence_evaluation_window]
            if len(acceptence_indx_over_window) < 2: # 1 or 0 acceptences within last window
            ↪  implies convergence
                print("converged")
                done = True
```

```
                return done
            else:  # caclulate diffrence between max and min over window
                earliest_acceptence_indx_over_window = acceptence_indx_over_window[0]
                best_accepted_value_over_recent_window = \
                    np.max(self.accepted_objective_history_array
                           [earliest_acceptence_indx_over_window:, 0])
                worst_accepted_value_over_recent_window = \
                    np.min(self.accepted_objective_history_array
                           [earliest_acceptence_indx_over_window:, 0])

                if best_accepted_value_over_recent_window -
                ↪  worst_accepted_value_over_recent_window < \
                        self.convergence_min_improvement:
                    print("converged")
                    done = True
                    return done
        # check if max iter has been reached
        if self.objective_function_evaluation_count >=
        ↪  self.objective_function_evaluation_max_count:
            done = True
        else:
            done = False
        return done

    def update_archive(self, x_new, objective_new):
        function_archive = [f_archive for x_archive, f_archive in self.archive]
        dissimilarity = [np.sqrt((x_archive - x_new).T @ (x_archive - x_new)) for x_archive,
        ↪  f_archive in self.archive]
        if min(dissimilarity) > self.archive_minimum_acceptable_dissimilarity:  # dissimilar to
        ↪  all points
            if len(self.archive) < self.archive_maximum_length:  # archive not full
                self.archive.append((x_new, objective_new))
            else:  # if archive is full
                if objective_new < min(function_archive):
                    self.archive[int(np.argmax(function_archive))] = (x_new, objective_new)  #
                    ↪  replace worst solution
        else:    # new solution is close to another
            if objective_new < min(function_archive):   # objective is lowest yet
                most_similar_indx = int(np.argmin(dissimilarity))
                self.archive[most_similar_indx] = (x_new, objective_new)  # replace most
                ↪  similar value
            else:
                similar_and_better = np.array([dissimilarity[i] <
                ↪  self.archive_similar_dissimilarity and \
                                       function_archive[i] > objective_new
                                       for i in range(len(self.archive))])
                if True in similar_and_better:
                    self.archive[np.where(similar_and_better == True)[0][0]] = (x_new,
                    ↪  objective_new)
```

```
423         if self.objective_function_evaluation_count %
            ↪  (int(self.objective_function_evaluation_max_count/10)) == 0:
424             # sometimes one value can like between 2 others, causing similarity even with the
                ↪  above loop
425             # clean_archive fixes this
426             # only need to do very rarely
427             self.clean_archive()
428
429
430     def clean_archive(self):
431         for x_new, y in self.archive:
432             dissimilarity = [np.sqrt((x_archive - x_new).T @ (x_archive - x_new)) for
                ↪  x_archive, f_archive in
433                             self.archive]
434             indxs_to_remove = np.where((np.array(dissimilarity) <
                ↪  self.archive_minimum_acceptable_dissimilarity) &
435                                        (self.archive_f > y))  # remove values that are close,
                                        ↪  with lower objectives
436             indxs_to_remove = indxs_to_remove[0]
437             if len(indxs_to_remove) > 0:
438                 for i, indx_to_remove in enumerate(indxs_to_remove):
439                     # deletions changes indexes so we have to adjust by i each time
440                     del(self.archive[indx_to_remove - i])
441
442     # often it was conventient to store values in lists
443     # however after the optimisation it is more convenient to have
444     # them as arrays, the below property methods are therefore given
445     @property
446     def temperature_history_array(self):
447         return np.array(self.temperature_history)
448
449     @property
450     def archive_x(self):
451         return np.interp(np.array([x_archive for x_archive, f_archive in self.archive]), [-1,
            ↪  1], self.x_bounds)
452
453     @property
454     def archive_f(self):
455         return np.array([f_archive for x_archive, f_archive in self.archive])
456
457     @property
458     def accepted_objective_history_array(self):
459         return np.array(self.accepted_objective_history)
460
461     @property
462     def accepted_x_history_array(self):
463         return np.interp(np.array(self.accepted_x_history), [-1, 1], self.x_bounds)
464
465     @property
```

```
466     def objective_history_array(self):
467         return np.array(self.objective_history)
468 
469     @property
470     def step_size_matrix_history_array(self):
471         return np.array(self.step_size_matrix_history)
472 
473     @property
474     def step_size_update_locations_array(self):
475         return np.array(self.step_size_update_locations)
476 
477     @property
478     def probability_of_acceptance_history_array(self):
479         return np.array(self.probability_of_acceptance_history)
480 
481     @property
482     def x_history_array(self):
483         return np.interp(np.array(self.x_history), [-1, 1], self.x_bounds)
484 
485     @property
486     def alpha_history_array(self):
487         return np.array(self.alpha_history)
488 
489     @property
490     def eigenvalue_eigenvector_history(self):
491         theta_history = []
492         eigen_values_history = []
493         for i in range(self.step_size_matrix_history_array.shape[0]):
494             step_size_matrix = self.step_size_matrix_history_array[i, :, :]
495             eigenvalues, eigenvectors = np.linalg.eig(step_size_matrix)
496             thetas = np.arccos(np.eye(self.x_length) @ eigenvectors)
497             min_thetas = np.min(thetas, axis=0)
498             order = np.argsort(-eigenvalues)
499             eigen_values_history.append(list(eigenvalues[order]))
500             theta_history.append(list(min_thetas[order]))
501         return np.array(eigen_values_history), np.array(theta_history)
502 
503 if __name__ == "__main__":
504     # simple example run with rana 5D function
505     np.random.seed(0)
506 
507     from rana import rana_func
508 
509     configuration = {"pertubation_method": "simple",
510                      "x_length": 5,
511                      "x_bounds": (-500, 500),
512                      "annealing_schedule": "simple_exponential_cooling",
513                      "objective_function": rana_func,
514                      "maximum_archive_length": 100,
```

```
                    "archive_minimum_acceptable_dissimilarity": 0.2,
                    "maximum_markov_chain_length": 50,
                    "maximum_function_evaluations": 10000,
                    "step_size_initialisation_fraction_of_range": 0.1,
                    "bound_enforcing_method": "not_clipping",
                    "cholesky_path_length": 5,
                    }
    np.random.seed(3)
    rana_2d_chol = SimulatedAnnealing(**configuration)
    x_result_chol, objective_result_chol = rana_2d_chol.run()
    print(f"x_result = {x_result_chol} \n objective_result = {objective_result_chol} \n "
          f"number of function evaluations =
          ↪  {rana_2d_chol.objective_function_evaluation_count}")
    print(f"best objective result {rana_2d_chol.objective_history_array.min()}")
```

6.2 Evolution Strategies code

```
import numpy as np


class EvolutionStrategy:
    """
    x is transformed from the original bounds, to be bounded by (-1, 1)
    Before outputs are given x is re-transformed back into the original form
    """
    def __init__(self, x_length, x_bounds, objective_function,
    ↪  archive_minimum_acceptable_dissimilarity=0.1,
                 parent_number=10,
                 selection_method="standard_mew_comma_lambda", mutation_method = "simple",
                 recombination_method="global",
                 termination_min_abs_difference=1e-6,
                 maximum_archive_length=None, objective_count_maximum=10000,
                 mutation_covariance_initialisation_fraction_of_range=0.01,
                 standard_deviation_clipping_fraction_of_range = 0.05,
                 bound_enforcing_method="not_clipping",
                 child_to_parent_ratio=7):
        self.x_length = x_length
        self.x_bounds = x_bounds
        self.bound_enforcing_method = bound_enforcing_method
        self.x_range = 2  # after being transformed to be between -1 and 1
        self.objective_function_raw = objective_function
        self.selection_method = selection_method
        self.mutation_method = mutation_method
        self.recombination_method = recombination_method
        self.termination_min_abs_difference = termination_min_abs_difference

        # prevent standard deviations from becoming too large (clip relative to size of range)
```

```
        self.standard_deviation_clipping_fraction_of_range =
        ↪  standard_deviation_clipping_fraction_of_range
        self.parent_number = parent_number
        self.offspring_number = parent_number * child_to_parent_ratio

        if mutation_method == "complex":
            # Mutation parameters - taken from slides, recommended by (Schwefel 1987)
            self.mutation_tau = 1/np.sqrt(2*np.sqrt(self.x_length))
            self.mutation_tau_dash = 1/np.sqrt(2*self.x_length)
            self.mutation_Beta = 0.0873
            self.offspring_mutation_standard_deviations = \
                np.ones((self.offspring_number, self.x_length)) * \
                mutation_covariance_initialisation_fraction_of_range * self.x_range
            self.parent_mutation_standard_deviations = \
                np.ones((self.parent_number, self.x_length)) * \
                mutation_covariance_initialisation_fraction_of_range * self.x_range

            self.offspring_rotation_matrices = np.broadcast_to(np.eye(self.x_length),
            ↪  (self.offspring_number, self.x_length, self.x_length))

            self.make_covariance_matrix()
            # just slice children for initialisation
            self.parent_rotation_matrices =
            ↪  self.offspring_rotation_matrices[0:self.parent_number, :, :]
            self.parent_covariance_matrices =
            ↪  self.offspring_covariance_matrices[0:self.parent_number, :, :]


        elif mutation_method == "simple":
            self.standard_deviation_simple =
            ↪  mutation_covariance_initialisation_fraction_of_range*self.x_range

        elif mutation_method == "diagonal":
            self.mutation_tau = 1 / np.sqrt(2 * np.sqrt(self.x_length))
            self.mutation_tau_dash = 1 / np.sqrt(2 * self.x_length)
            self.offspring_mutation_standard_deviations = \
                np.ones((self.offspring_number, self.x_length)) * \
                mutation_covariance_initialisation_fraction_of_range * self.x_range
            self.parent_mutation_standard_deviations = np.ones((self.parent_number,
            ↪  self.x_length)) * mutation_covariance_initialisation_fraction_of_range *
            ↪  self.x_range

        # initialise parents and offspring
        # zeros aren't ever used, just specifies the shapes of the arrays
        self.parents = np.zeros((self.parent_number, self.x_length))
        self.parent_objectives = np.zeros(self.parent_number)
        self.offspring = np.zeros((self.offspring_number, self.x_length))
        self.offspring_objectives = np.zeros(self.offspring_number)

```

```
72
73          # initialise archive and parameters determining how archive is managed
74          self.archive = []   # list of (x, objective value) tuples
75          self.archive_maximum_length = maximum_archive_length    # If none then don't store, as
            ↪  slows program down slightly
76          self.archive_minimum_acceptable_dissimilarity =
            ↪  archive_minimum_acceptable_dissimilarity
77          self.archive_similar_dissimilarity = archive_minimum_acceptable_dissimilarity
78
79          # initialise histories and counters
80          # these are useful for inspecting the performance of the algorithm after a run
81          # and are used within the program (e.g. markov chain length)
82          self.parent_objective_history = []
83          self.parent_x_history = []
84          self.parent_standard_deviation_history = []
85          self.offspring_objective_history = []
86          self.offspring_x_history = []
87          self.parent_covariance_determinant_history = []
88          self.offspring_covariance_determinant_history = []
89          self.objective_function_evaluation_count = 0  # initialise
90          self.generation_number = 0
91          self.objective_function_evaluation_max_count = objective_count_maximum
92
93      def objective_function(self, x):
94          """
95          Wrapper for the objective function calls, adding some extra functionality
96          """
97          # increment by one everytime objective function is called
98          self.objective_function_evaluation_count += 1
99          # interpolation done here to pass the objective function x correctly interpolated
100         x_interp = np.interp(x, [-1, 1], self.x_bounds)
101         result = self.objective_function_raw(x_interp)
102         return result
103
104     def run(self):
105         """
106         This function run's the major steps of the Evolution Strategy algorithm
107         # major steps in the algorithm are surrounded with a #******************************
108         # other parts of this function are more organisation (e.g. storing histories)
109         """
110         self.initialise_random_population()
111         while True:  # loop until termination criteria is reached
112             self.generation_number += 1
113             # *************************   Selection ******************
114             self.select_parents()
115             # ************************************************************
116             if self.archive_maximum_length is not None: # if archive is None, then don't store
117                 for x, objective in zip(self.parents, self.parent_objectives):  # update
                    ↪  archive
```

```
                    self.update_archive(x, objective)
            self.parent_objective_history.append(self.parent_objectives)
            self.parent_x_history.append(self.parents)
            if self.mutation_method == "diagonal":
                self.parent_standard_deviation_history.append\
                    (self.parent_mutation_standard_deviations)
                self.parent_covariance_determinant_history.append(
                    np.prod(self.parent_mutation_standard_deviations, axis=1))
                self.offspring_covariance_determinant_history.append(
                    np.prod(self.offspring_mutation_standard_deviations, axis=1))
            elif self.mutation_method == "complex":
                self.parent_standard_deviation_history.append(
                    self.parent_mutation_standard_deviations)
                self.parent_covariance_determinant_history.append(
                    np.linalg.det(self.parent_covariance_matrices))
                self.offspring_covariance_determinant_history.append(
                    np.linalg.det(self.offspring_covariance_matrices))

            # *************  Check for convergence/termination      ************
            # ensure termination before 10000 iterations
            if self.objective_function_evaluation_count >
             ↪  self.objective_function_evaluation_max_count-self.offspring_number:
                print("max total iterations")
                break

            # termination criteria
            if max(self.parent_objectives) - min(self.parent_objectives) <
             ↪  self.termination_min_abs_difference:
                print("converged")
                break
            # ************************************************************
            # *********************   Create Offspring *******************
            # I.e. perform recombination and mutation to create offspring
            self.create_new_offspring()
            # ************************************************************
            self.offspring_objective_history.append(self.offspring_objectives)
            self.offspring_x_history.append(self.offspring)

        best_x = self.parents[np.argmin(self.parent_objectives), :]
        best_objective = min(self.parent_objectives)
        return np.interp(best_x, [-1, 1], self.x_bounds), best_objective

    def initialise_random_population(self):
        self.offspring = np.random.uniform(low=-1, high=1, size=(self.offspring_number,
        ↪  self.x_length))
        self.offspring_objectives =
        ↪  np.squeeze(np.apply_along_axis(func1d=self.objective_function, arr=self.offspring,
        ↪  axis=1))
        if self.selection_method == "elitist":  # require pool including parents for
        ↪  select_parents function in this case
```

```
            self.parents = np.random.uniform(low=-1, high=1,
                                             size=(self.parent_number, self.x_length))
            self.parent_objectives = np.apply_along_axis(func1d=self.objective_function,
            ↪  arr=self.parents, axis=1)

    def select_parents(self):
        if self.selection_method == "standard_mew_comma_lambda":
            # choose top values in linear time
            # np.argpartition doesn't sort top values amongst themselves so is compuationally
            ↪  faster
            pool_objectives = self.offspring_objectives
            pool = self.offspring
            if self.mutation_method == "diagonal":
                pool_standard_deviations = self.offspring_mutation_standard_deviations
            if self.mutation_method == "complex":
                pool_standard_deviations = self.offspring_mutation_standard_deviations
                pool_rotation_matrices = self.offspring_rotation_matrices
                pool_covariance_matrices = self.offspring_covariance_matrices
        else:
            assert self.selection_method == "elitist"
            # create pool selcted from
            pool_objectives = np.zeros(self.parent_number + self.offspring_number)
            pool_objectives[0:self.offspring_number] = self.offspring_objectives
            pool_objectives[self.offspring_number:] = self.parent_objectives
            pool = np.zeros((self.offspring_number + self.parent_number, self.x_length))
            pool[0:self.offspring_number, :] = self.offspring
            pool[self.offspring_number:, :] = self.parents
            if self.mutation_method == "diagonal":
                pool_standard_deviations = np.zeros((self.offspring_number +
                ↪  self.parent_number, self.x_length))
                pool_standard_deviations[0:self.offspring_number, :] =
                ↪  self.offspring_mutation_standard_deviations
                pool_standard_deviations[self.offspring_number:, :] =
                ↪  self.parent_mutation_standard_deviations
            if self.mutation_method == "complex":
                pool_standard_deviations = np.zeros((self.offspring_number +
                ↪  self.parent_number, self.x_length))
                pool_standard_deviations[0:self.offspring_number, :] =
                ↪  self.offspring_mutation_standard_deviations
                pool_standard_deviations[self.offspring_number:, :] =
                ↪  self.parent_mutation_standard_deviations
                pool_rotation_matrices = np.zeros((self.offspring_number + self.parent_number,
                ↪  self.x_length, self.x_length))
                pool_rotation_matrices[0:self.offspring_number, :, :] =
                ↪  self.offspring_rotation_matrices
                pool_rotation_matrices[self.offspring_number:, :, :] =
                ↪  self.parent_rotation_matrices
                pool_covariance_matrices = np.zeros((self.offspring_number +
                ↪  self.parent_number, self.x_length, self.x_length))
```

```
                pool_covariance_matrices[0:self.offspring_number, :, :] =
                ↪ self.offspring_covariance_matrices
                pool_covariance_matrices[self.offspring_number:, :, :] =
                ↪ self.parent_covariance_matrices


        new_parent_indxs = np.argpartition(pool_objectives,
        ↪ self.parent_number)[:self.parent_number]
        self.parents = pool[new_parent_indxs, :]
        self.parent_objectives = pool_objectives[new_parent_indxs]

        if self.mutation_method == "diagonal":
            self.parent_mutation_standard_deviations =
            ↪ pool_standard_deviations[new_parent_indxs, :]
        elif self.mutation_method == "complex":
            self.parent_mutation_standard_deviations =
            ↪ pool_standard_deviations[new_parent_indxs, :]
            self.parent_rotation_matrices = pool_rotation_matrices[new_parent_indxs, :, :]
            self.parent_covariance_matrices = pool_covariance_matrices[new_parent_indxs, :, :]


    def create_new_offspring(self):
        """
        Recombination and Mutation
        """
        #****************   Recombination   *******************************
        # global discrete recombination for control parameters
        # global intermediate recombination for stratergy parameters
        if self.recombination_method == "global":
            # for each element in each child, inherit from a random parent
            child_recombination_indxs = np.random.choice(self.parent_number, replace=True,
                                        size = (self.offspring_number, self.x_length))
            offspring_pre_mutation = self.parents[child_recombination_indxs,
            ↪ np.arange(self.x_length)]
            if self.mutation_method == "diagonal" or self.mutation_method == "complex":
                child_stratergy_recombination_indxs = np.random.choice(self.parent_number,
                ↪ replace=True,
                                        size=(self.offspring_number,  self.x_length, 2))
                offspring_pre_mutation_standard_deviation = \
                    0.5*self.parent_mutation_standard_deviations[
                        child_stratergy_recombination_indxs[:, :, 0], np.arange(self.x_length)]
                        ↪ +\
                    0.5*self.parent_mutation_standard_deviations[
                        child_stratergy_recombination_indxs[:, :, 1], np.arange(self.x_length)]
                if self.mutation_method == "complex":
                    child_stratergy_recombination_indxs = \
                        np.random.choice(self.parent_number, replace=True,
                        ↪ size=(self.offspring_number*self.x_length*self.x_length, 2))
                    slices1 = np.broadcast_to(np.arange(self.x_length)[np.newaxis, :],
                    ↪ (self.x_length, self.x_length)).flatten()
```

```
                slices1 = np.broadcast_to(slices1[np.newaxis, :], (self.offspring_number,
                ↪  self.x_length**2)).flatten()
                slices2 = np.broadcast_to(np.arange(self.x_length)[:, np.newaxis],
                ↪  (self.x_length, self.x_length)).flatten()
                slices2 = np.broadcast_to(slices2[np.newaxis, :], (self.offspring_number,
                ↪  self.x_length**2)).flatten()
                offspring_pre_mutation_rotation_matrix = \
                    np.reshape(0.5 * self.parent_rotation_matrices[
                        child_stratergy_recombination_indxs[:, 0], slices1,
                        slices2 ] + \
                    0.5 * self.parent_rotation_matrices[
                        child_stratergy_recombination_indxs[:, 1], slices1,
                        slices2], (self.offspring_number, self.x_length, self.x_length))

        #****************   Mutation   ******************************
        if self.mutation_method == "simple":
            u_random_sample = np.random.normal(loc=0, scale=self.standard_deviation_simple,
                                               size=offspring_pre_mutation.shape)
            x_new = offspring_pre_mutation + u_random_sample
            if self.bound_enforcing_method == "clipping":
                x_new = np.clip(x_new, -1, 1)
            else:
                while np.max(x_new) > 1 or np.min(x_new) < -1:
                    indxs_breaking_bounds = np.where((x_new > 1) + (x_new < -1) == 1)
                    u_random_sample = np.random.normal(loc=0,
                    ↪  scale=self.standard_deviation_simple,
                                                       size=indxs_breaking_bounds[0].size)
                    x_new[indxs_breaking_bounds ] =
                    ↪  offspring_pre_mutation[indxs_breaking_bounds ] + u_random_sample


        elif self.mutation_method == "diagonal":  # non spherical covariance
            self.offspring_mutation_standard_deviations = \
                offspring_pre_mutation_standard_deviation * \
                np.exp(self.mutation_tau_dash*np.broadcast_to(
                            np.random.normal(0, 1, size=(self.offspring_number, 1)),
                             ↪  self.offspring_mutation_standard_deviations.shape)
                           + self.mutation_tau*np.random.normal(0, 1,
                            ↪  size=self.offspring_mutation_standard_deviations.shape))
            self.offspring_mutation_standard_deviations = \
                np.clip(self.offspring_mutation_standard_deviations,
                        1e-8, self.standard_deviation_clipping_fraction_of_range*self.x_range)

            u_random_sample = np.random.normal(loc=0,
            ↪  scale=self.offspring_mutation_standard_deviations,
                                               size=offspring_pre_mutation.shape)
            x_new = offspring_pre_mutation + u_random_sample
            if self.bound_enforcing_method == "clipping":
                x_new = np.clip(x_new, -1, 1)
```

```
            else:
                while np.max(x_new) > 1 or np.min(x_new) < -1:
                    indxs_breaking_bounds = np.where((x_new > 1) + (x_new < -1) == 1)
                    u_random_sample = \
                        np.random.normal(loc=0,
                        ↪  scale=self.offspring_mutation_standard_deviations
                        [indxs_breaking_bounds],size=indxs_breaking_bounds[0].size)
                    x_new[indxs_breaking_bounds] =
                    ↪  offspring_pre_mutation[indxs_breaking_bounds] + u_random_sample

        if self.mutation_method == "complex":
            self.offspring_mutation_standard_deviations = \
                offspring_pre_mutation_standard_deviation  * \
                np.exp(self.mutation_tau_dash*np.broadcast_to(
                            np.random.normal(0, 1, size=(self.offspring_number, 1)),
                            ↪  self.offspring_mutation_standard_deviations.shape)
                           + self.mutation_tau*np.random.normal(0, 1,
                           ↪  size=self.offspring_mutation_standard_deviations.shape))

            self.offspring_mutation_standard_deviations = \
                np.clip(self.offspring_mutation_standard_deviations, 1e-8,
                self.standard_deviation_clipping_fraction_of_range * self.x_range)

            self.offspring_rotation_matrices = offspring_pre_mutation_rotation_matrix +
            ↪  self.mutation_Beta * np.random.normal(0, 1,
            ↪  size=(self.offspring_rotation_matrices.shape))
            for i in range(self.offspring_number):
                self.offspring_rotation_matrices[i, :, :] =
                ↪  np.tril(self.offspring_rotation_matrices[i, :, :], k=-1) - np.tril(
                    self.offspring_rotation_matrices[i, :, :], k=-1).T      # make symmetric
            self.make_covariance_matrix()
            for i in range(self.offspring_number):
                covariance_matrix =
                ↪  self.make_positive_definate(self.offspring_covariance_matrices[i, :, :])
                self.offspring[i, :] = offspring_pre_mutation[i, :] +
                ↪  np.random.multivariate_normal(mean=np.zeros(self.x_length),
                ↪  cov=covariance_matrix)
                if self.bound_enforcing_method == "clipping":
                    self.offspring[i, :]  =  np.clip(self.offspring[i, :] , -1, 1)
                else:
                    while np.max(self.offspring[i, :]) > 1 or np.min(self.offspring[i, :]) <
                    ↪  -1:
                        self.offspring[i, :] = offspring_pre_mutation[i, :] +
                        ↪  np.random.multivariate_normal(mean=np.zeros(self.x_length),
                        ↪  cov=covariance_matrix)
        if self.mutation_method != "complex":
            self.offspring = x_new
        self.offspring_objectives = np.squeeze(
            np.apply_along_axis(func1d=self.objective_function, arr=self.offspring, axis=1))
```

```
316
317     def make_positive_definate(self, matrix, i=1):
318         try:
319             np.linalg.cholesky(matrix)
320             return matrix
321         except:
322             if i > 10:
323                 raise Exception("matrix unable to be made positive definate")
324             matrix += np.eye(self.x_length) * 1e-6 * 10**i
325             return self.make_positive_definate(matrix, i=i+1)
326
327
328     def update_archive(self, x_new, objective_new):
329         if len(self.archive) == 0:  # if empty then initialise with the first value
330             self.archive.append((x_new, objective_new))
331         function_archive = [f_archive for x_archive, f_archive in self.archive]
332         dissimilarity = [np.sqrt((x_archive - x_new).T @ (x_archive - x_new)) for x_archive,
        ↪  f_archive in self.archive]
333         if min(dissimilarity) > self.archive_minimum_acceptable_dissimilarity:  # dissimilar to
        ↪  all points
334             if len(self.archive) < self.archive_maximum_length:  # archive not full
335                 self.archive.append((x_new, objective_new))
336             else:  # if archive is full
337                 if objective_new < min(function_archive):
338                     self.archive[int(np.argmax(function_archive))] = (x_new, objective_new)  #
                    ↪  replace worst solution
339         else:  # new solution is close to another
340             if objective_new < min(function_archive):  # objective is lowest yet
341                 most_similar_indx = int(np.argmin(dissimilarity))
342                 self.archive[most_similar_indx] = (x_new, objective_new)  # replace most
                ↪  similar value
343             else:
344                 similar_and_better = np.array([dissimilarity[i] <
                ↪  self.archive_similar_dissimilarity and \
345                                                function_archive[i] > objective_new
346                                                for i in range(len(self.archive))])
347                 if True in similar_and_better:
348                     self.archive[np.where(similar_and_better == True)[0][0]] = (x_new,
                    ↪  objective_new)
349         if self.generation_number % 10 == 0:
350             # sometimes one value can like between 2 others, causing similarity even with the
            ↪  above loop
351             # clean_archive fixes this
352             # only need to do very rarely
353             # slows down program a lot, so only perform when we need to visualise 2D problem
354             self.clean_archive()
355
356     def clean_archive(self):
357         # first remove repeats
```

```
        for x_new, y in self.archive:
            dissimilarity = [np.sqrt((x_archive - x_new).T @ (x_archive - x_new)) for
            ↪  x_archive, f_archive in
                             self.archive]
            indxs_to_remove = np.where(np.array(dissimilarity) ==0)  # remove values that are
            ↪  close, with lower objectives
            indxs_to_remove = indxs_to_remove[0]
            if len(indxs_to_remove) > 0:
                indxs_to_remove = indxs_to_remove[1:]  # remove all but the first copy
                for i, indx_to_remove in enumerate(indxs_to_remove):
                    # deletions changes indexes so we have to adjust by i each time
                    del (self.archive[indx_to_remove - i])

        # then remove overly similar
        for x_new, y in self.archive:
            dissimilarity = [np.sqrt((x_archive - x_new).T @ (x_archive - x_new)) for
            ↪  x_archive, f_archive in
                             self.archive]
            indxs_to_remove = np.where((np.array(dissimilarity) <
            ↪  self.archive_minimum_acceptable_dissimilarity) &
                                       (self.archive_f > y))  # remove values that are close,
                                       ↪  with lower objectives
            indxs_to_remove = indxs_to_remove[0]
            if len(indxs_to_remove) > 0:
                for i, indx_to_remove in enumerate(indxs_to_remove):
                    # deletions changes indexes so we have to adjust by i each time
                    del (self.archive[indx_to_remove - i])

    def make_covariance_matrix(self):
        sigma_i = np.zeros((self.offspring_number, self.x_length, self.x_length))
        sigma_j = np.zeros((self.offspring_number, self.x_length, self.x_length))
        for offspring_number in range(self.offspring_number):
            stds = self.offspring_mutation_standard_deviations[offspring_number, :]
            sigma_i[offspring_number, np.arange(self.x_length), :] = stds
            sigma_j[offspring_number, :, np.arange(self.x_length)] = stds
        self.offspring_covariance_matrices = np.tan(2 * self.offspring_rotation_matrices) *
        ↪  (sigma_i**2 - sigma_j**2) * 1/2
        self.offspring_covariance_matrices = np.clip(self.offspring_covariance_matrices,
        ↪  -np.minimum(sigma_i, sigma_j), np.minimum(sigma_i, sigma_j))
        self.offspring_covariance_matrices[:, np.arange(self.x_length),
        ↪  np.arange(self.x_length)] = self.offspring_mutation_standard_deviations

    # often it was conventient to store values in lists
    # however after the optimisation it is more convenient to have
    # them as arrays, the below property methods are therefore given
    @property
    def parent_objective_history_array(self):
        return np.array(self.parent_objective_history)

```

```
    @property
    def offspring_objective_history_array(self):
        return np.array(self.offspring_objective_history)

    @property
    def parent_standard_deviation_history_array(self):
        return np.array(self.parent_standard_deviation_history)

    @property
    def parent_covariance_determinant_history_array(self):
        return np.array(self.parent_covariance_determinant_history)

    @property
    def offspring_covariance_determinant_history_array(self):
        return np.array(self.offspring_covariance_determinant_history)
    @property
    def offspring_x_history_array(self):
        return np.interp(np.array(self.offspring_x_history), [-1, 1], self.x_bounds)
    @property
    def parent_x_history_array(self):
        return np.interp(np.array(self.parent_x_history), [-1, 1], self.x_bounds)

    @property
    def archive_x(self):
        return np.interp(np.array([x_archive for x_archive, f_archive in self.archive]), [-1,
        ↪  1], self.x_bounds)

    @property
    def archive_f(self):
        return np.array([f_archive for x_archive, f_archive in self.archive])



if __name__ == "__main__":
    # example run on the 5 D rana problem
    from rana import rana_func
    Comp_config = {"objective_function": rana_func,
                   "x_bounds": (-500, 500),
                   "x_length": 5,
                   "parent_number": 10,
                   "child_to_parent_ratio": 7,
                   "bound_enforcing_method": "not_clipping",
                   "selection_method": "standard_mew_comma_lambda",
                   "standard_deviation_clipping_fraction_of_range": 0.01,
                   "mutation_covariance_initialisation_fraction_of_range": 0.01,
                   "mutation_method": "complex",
                   "termination_min_abs_difference": 1e-6,
                   "maximum_archive_length": 20}
    random_seed = 1
```

```
    np.random.seed(random_seed)
    x_max = 500
    x_min = -x_max
    evo_comp = EvolutionStrategy(**Comp_config)
    x_result, objective_result = evo_comp.run()
    print(f"x_result = {x_result} \n objective_result = {objective_result}\n\n\n\
          number of objective_evaluations is {evo_comp.objective_function_evaluation_count}\
          number of generations is {evo_comp.generation_number}")
```

6.3 Random search code

```
import numpy as np
from rana import rana_func
class random_search:
    def __init__(self, x_length, x_bounds= (-500, 500), objective_function=rana_func,
    ↪  n_evaluations=10000):
        self.x_length = x_length
        self.x_bounds = x_bounds
        self.objective_function = objective_function
        self.n_evaluations = n_evaluations

    def run(self):
        self.all_points = np.random.uniform(low=self.x_bounds[0], high=self.x_bounds[1],
        ↪  size=(self.n_evaluations, self.x_length))
        self.objectives = np.apply_along_axis(func1d=self.objective_function,
        ↪  arr=self.all_points, axis=1)
        return self.objectives.min(), self.all_points[np.argmin(self.objectives), :]
```