4M17 Coursework Assignment 2

Practical Optimisation

Candidate Number: 5746G

January 10, 2021

1 Introduction

A general *optimisation problem* has the form:

$$\begin{array}{ll} \text{minimize} & f_0(\mathbf{x}) \\ \text{subject to} & f_i(\mathbf{x}) \leq b_i, \quad i = 1, \ldots, m \end{array} \tag{1}$$

where the vector $\mathbf{x} \in \mathbb{R}^n$ is the n-dimensional *optimisation variable* of the problem, the function $f_0 : \mathbb{R}^n \to \mathbb{R}$ is the *objective function*, and the functions $f_i : \mathbb{R}^n \to \mathbb{R}$, for $i = 1, \ldots, m$, are the *constraint functions*. The mathematical procedure of optimisation involves finding the *optimal solution* $\mathbf{x}^*$ which has the smallest object value among all vectors which satisfy the constraints. That is to say, for any $\mathbf{x}$ with $f_i(\mathbf{x}) \leq b_i$ for $i = 1, \ldots, m$, then $f_0(\mathbf{x}) \geq f_0(\mathbf{x}^*)$ (Boyd & Vandenberghe 2004).

The first of the two 4M17 coursework assignments addressed the topic of *convex optimisation*, a subset of optimisation problems in which the objective and constraint functions are convex, meaning they satisfy the inequality:

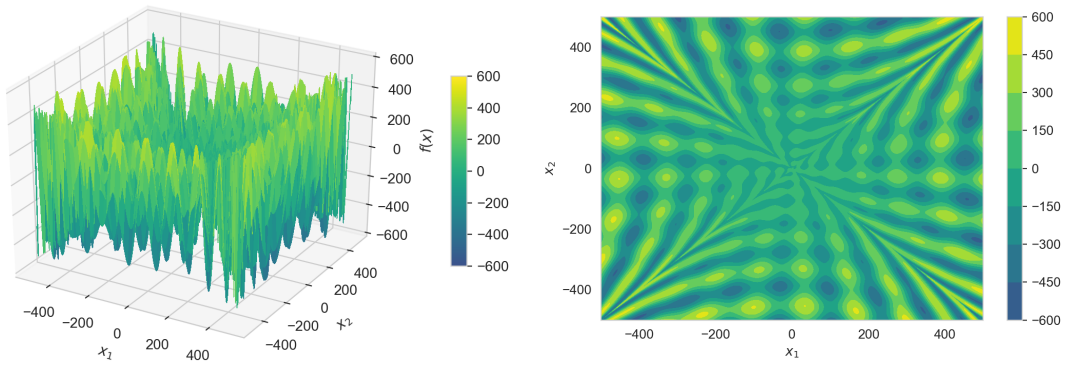
$$f_i(\alpha\mathbf{x} + \beta\mathbf{y}) \leq \alpha f_i(\mathbf{x}) + \beta f_i(\mathbf{y}), \tag{2}$$

for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, and all $\alpha, \beta \in \mathbb{R}$ with $\alpha + \beta = 1$, $\alpha, \beta \geq 0$. The advantage of working with convex problems is that they can be solved very efficiently and reliably using established numerical techniques, such as interior-point methods, often yielding polynomial solve times (Nesterov & Nemirovskii 1994). However, many real world optimisation problems are non-convex, and as such require a different category of optimisation techniques.

This report investigates two algorithms belonging to the class of *stochastic optimisation methods*: *simulated annealing* (SA) and *evolutionary strategies* (ESs). Both of these methods draw inspiration from real-world processes. There is a deep connection between SA and the process of *annealing* in metallurgy, a technique in which a metal is heated and its cooling controlled to produce a state of high order and very low energy (Kirkpatrick et al. 1982). ESs are a class of probabilistic search algorithms based on the *Darwinian model of evolution* (Darwin 1859, Schwefel 1977).

These heuristic approaches provide a means of iteratively searching over the search-space more efficiently than a purely *exhaustive search*. Probabilistic in nature, they introduce a degree of randomness into the search-process to accelerate the procedure of solving highly *multimodal nonlinear* mathematical optimisation problems. *Deterministic gradient methods* alone are unsuitable for problems with multimodalities, becoming 'trapped' in *local minima*. Stochastic algorithms, on the other hand, accept some changes which increase the objective function, without the need for first or second-order *derivative information*, thus overcoming this issue.

In general, these algorithms guarantee only to attain a locally optimum solution, which in some cases may be the *global optimum* (Schneider & Kirkpatrick. 2006). The process of *tuning* 'out of the box' algorithms to a problem specific function is explored, and the effects of varying parameter values on each algorithm discussed.



(a) **(b)**

Figure 1: Panel (a) plots surface of the surface of Rana's Function for $n = 2$. Panel (b) shows a visualisation of the contours, evenly spaced at intervals of 150.

2 The Problem

The problem investigated is *Rana's Function* (Whitley et al. 1995, 1996), an n-dimensional multi-modal optimisation problem with inequality constraints:

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{x}) = \sum_{i=1}^{n-1} x_i \cos\left(\sqrt{|x_{i+1} + x_i + 1|}\right) \sin\left(\sqrt{|x_{i+1} - x_i + 1|}\right) \\ & \quad + (1 + x_{i+1}) \cos\left(\sqrt{|x_{i+1} - x_i + 1|}\right) \sin\left(\sqrt{|x_{i+1} + x_i + 1|}\right) \\ \text{subject to} \quad & -500 \leq x_i \leq 500, \quad i = 1, \ldots, n \end{aligned} \tag{3}$$

The report assesses the performance of the two stochastic algorithms on the 5-dimensional version of Rana's Function (5D-RF), with one further constraint that only 10,000 objective function evaluations are permitted for each execution of an algorithm.

Figures 1(a) and 1(b) visualise the surface and contours of the lower dimensional form of the problem for $n = 2$ (2D-RF). These plots reveal the extreme multimodality of the problem: the search-space is loitered with many local maxima and minima, resulting in the function being difficult in general to optimise.

Both plots in Figure 1 are generated using a *mesh-spacing* of 0.05, sub-dividing the input domain for each $x_i \in [-500, 500]$ into $500/0.05 = 10000$ discrete points. An *exhaustive search*, in which the 2D-RF is every evaluated point in the discretised search-space, requires 10000^2 evaluations of the 2D-RF. In general, for a mesh-spacing k, the domain of each x_i is sub-divided into $500/k$ discrete points. Given an n-dimensional optimisation problem, this brute-force method requires the objective function to be evaluated at $(500/k)^n$ points. The computational cost quickly becomes infeasible for increasing dimensionality; the number of iterations scales exponentially with the number of dimensions n. This undesirable trait, often referred to as 'the curse of dimensionality' (Taylor 1993), motivates the need for efficient, scalable search algorithms.

The 2D-RF is the only form of Rana's Function which can easily be visualised. It, therefore, provides a useful aid in representing search patterns followed by implementations of optimisation algorithms, in order to confirm they are operating as intended before being trialled on the higher dimensional 5D-RF.

3 Simulated Annealing

3.1 Outline of Implementation

A version of the simulated annealing (SA) algorithm was implemented in Python, code for which is given in Appendix A.1.

An *initial solution* $\mathbf{x}_0$ is generated by drawing from a uniform distribution over the entire search-space. Based on the suggestion by Kirkpatrick (1984), the initial temperature T_0 is calculated from an initial search to give an average probability $\chi_0 = 0.8$ of a solution that increases the objective function being accepted. Trial solutions are generated according to the formulation suggested by Parks (1990):

$$\mathbf{x}_{i+1} = \mathbf{x}_i + D\mathbf{u}, \tag{4}$$

where $\mathbf{u}$ is a vector of uniform random variables over $[-1, 1]$, and D is a diagonal matrix which specifies the maximum change allowed in each variable. Proposed solutions $\mathbf{x}_{i+1}$ which violate the inequality constraints of the optimisation problem in (3) are rejected immediately.

The elements of D are updated following each accepted changed according to the update rule:

$$D_{j+1} = (1 - \alpha)D_j + \alpha\omega R, \tag{5}$$

where the entries of the diagonal matrix R encode the magnitudes of the successful made to each element of $\mathbf{x}_i$:

$$R_{kk} = |D_{kk}u_k|, \tag{6}$$

and the *damping constant* α determines the rate at which update information is incorporated into D with *weighting* ω. Upper and lower limits D_u and D_l are placed on the values of the elements of D to prevent single values dominating or shrinking.

The *actual step size* $\bar{d}$ is given by:

$$\bar{d}^2 = \sum_{k=1}^{n} R_{kk}^2, \tag{7}$$

from which the probability of accepting the trial solution is calculated as:

$$p = \max\left\{1, \exp\left(-\frac{\delta f}{T\bar{d}}\right)\right\}, \tag{8}$$

meaning all decreases in the objective function are accepted, and any increases accepted with probability $\exp\left(-\frac{\delta f}{T\bar{d}}\right)$, provided the trial solution $\mathbf{x}_{i+1}$ is in the *feasible region*. Following the acceptance of a new solution, the *archive* of best solutions is updated. There is a maximum specified length L_k for the Markov chain at every temperature, from which a minimum number of acceptances is calculated as $\eta_{min} = 0.6L$. The temperature of the scheme is decremented following either L_k trials or η_{min} acceptances at a given temperature:

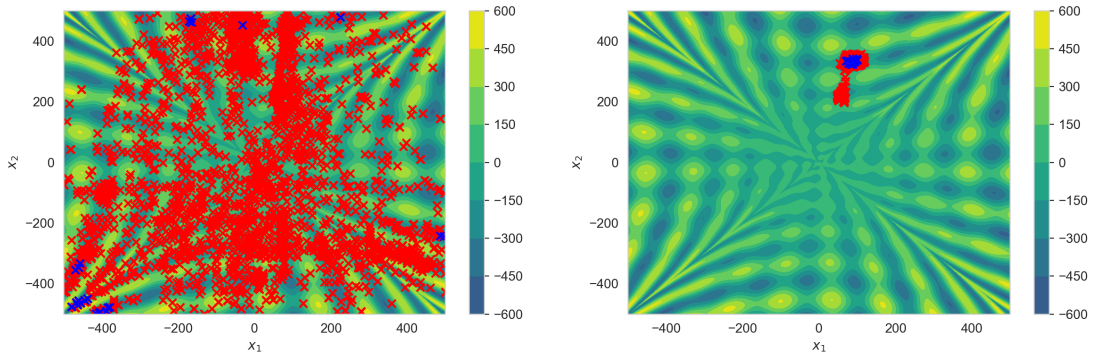
$$T_{k+1} = \alpha_k T_k, \tag{9}$$

where the factor $\alpha_k = 0.95$ for the *exponential cooling scheme* (ECS) proposed by Kirkpatrick et al. (1982), or:

$$\alpha_k = \max\left\{0.5, \exp\left(-\frac{0.7T_k}{\sigma_k}\right)\right\} \tag{10}$$

for the *adaptive cooling scheme* (ACS) outlined by Huang et al. (1986), where σ_k is the standard deviation of the objective function values accepted at temperature T_k.

A *restart* is performed following a specified number of iterations over which no new best solution is found. The algorithm is *terminated* after reaching the maximum number of permitted of objective function evaluations or ceasing to make progress, whichever comes first. The latter condition is defined as no new best solution being found in an entire Markov chain for a given temperature, alongside the *solution acceptance ratio* falling below a specified threshold $\chi_f = 0.01$.



(a) ECS **(b)** ACS

Figure 2: All accepted SA solutions (red) and final archived solutions (blue) for the ECS and ACS with $L_k = 100$.

3.2 Cooling Scheme and Maximum Markov Chain Length

The first parameters investigated were the choice of cooling[1] scheme, either exponential or adaptive, in conjunction with the maximum Markov chain length L_k. The temperature governs the balance between *exploration* of the search-space, and *exploitation* of good solutions. As the size of the temperature decrements is decreased, the probability that the algorithm terminates at the global optimum tends to 1 (Granville et al. 1994). However, infinitesimally small step sizes are practically infeasible, since the run-time of the algorithm also increases and quickly exceeds that of a purely exhaustive search. On the other hand, cooling too fast results in low probabilities of accepting steps which increase the objective function, increasing the likelihood of the optimiser becoming trapped in a local optimum. Thus, SA performance is conditional on selecting an appropriate annealing schedule. Finding an effective temperature decrement rule is often a problem specific task.

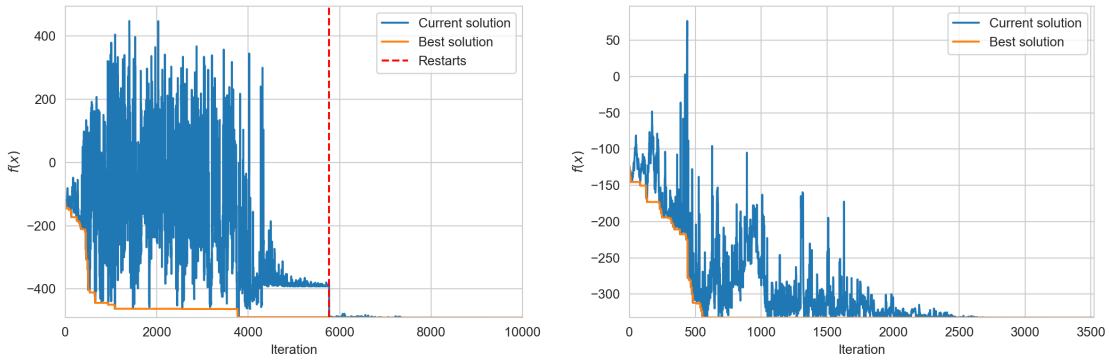
The reason for analysing the two parameters together is due to their interaction: the temperature is decremented after L_k iterations at the current temperature or after $\eta_{min} = 0.6L_k$ acceptances, whichever occurs first. A sufficient number of iterations are required at each temperature such that a *quasi-equilibrium* state is achieved. This is state is defined as the probability distribution of solutions $q(N, T_k)$ after N trials at temperature T_k being 'sufficiently close' to the stationary distribution $\pi(T_k)$ at T_k:

$$\left\| q(N, T_k) - \pi(T_k) \right\| < \epsilon, \tag{11}$$

where ϵ is a small, positive value. It has been shown by Aarts et al. (1997) that larger drops in temperature require longer-length Markov chains to reach the quasi-equilibrium state, and so require larger values of L_k. Hence, the optimal configuration of L_k for a given problem is dependent on the choice of cooling scheme. This notion is highlighted by Figure 2, which plots all accepted solutions and the final archived solutions for both the ECS and ACS. Despite using the same maximum chain length of $L_k = 100$, a very different distribution of solutions is obtained for each of the cooling schemes.

For the ECS, the annealer explored a significant portion of the search-space, solutions being generated for a wide range of (x_1, x_2) pairs. The dense bands of solutions, one between $x_1 = 0$ and $x_1 = 200$ spanning over the range of input values $x_2 > 0$, and another between $x_2 = -200$ and $x_2 = -400$ spanning over the range of input values $x_1 > 0$, correspond to samples accepted early on in the annealing process when the temperature was high. As the temperature cooled, the majority of samples were generated in the vicinity of the local minima, the algorithm settling on the regions where the function takes its lowest values. The 25 final archived solutions are positioned in similar such regions. This behaviour is verified by the variation of the evaluated objective function with SA

[1] In this report, the phrases *cooling* and *annealing* are used interchangeably. Both terms refer to the reduction of the temperature over the SA search.



(a) ECS **(b)** ACS

Figure 3: Evolution of the objective function as a function of SA iterations for both candidate cooling schemes, ECS and ACS, corresponding to the solutions obtained in Figures 2(a) and 2(b) respectively.

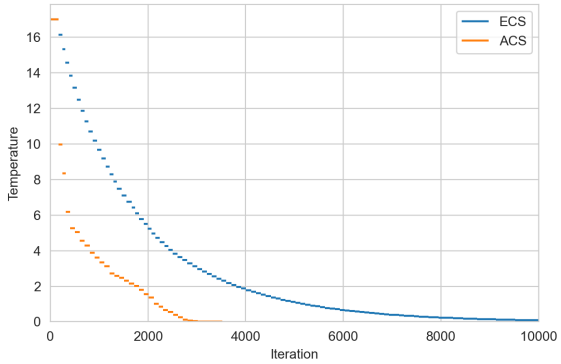


Figure 4: Variation of temperature with SA iterations for both candidate cooling schemes, ECS and ACS, corresponding to the solutions obtained in Figures 2(a) and 2(b) respectively.

iterations, a plot of which is shown in Figure 3(a). Early on in the process, when the temperature was high, the optimiser accepted a large number of solutions which increased the objective function, allowing it to navigate over the search-space. As the temperature was cooled, fewer objective function increases were accepted, and those which were took smaller magnitude. This is the typical search pattern taken by a functional SA algorithm, confirming the written code operated correctly on the 2D-RF.

In contrast to the ECS, the SA algorithm with the ACS explored only a very small proportion of the 2-dimensional search-space. Figure 4 plots the system temperature against SA iterations for both schemes, revealing that the ACS cooled significantly faster than the ECS. The search pattern of the adaptive annealing schedule is explained by the formulation for α_k given in eq. (10): beginning in the vicinity of a local optimum, the variance σ_k^2 of objective function values accepted was low for temperatures T_k early on in the optimisation procedure. This resulted in small corresponding values of α_k, and hence rapid initial cooling. This meant increases in the objective function were assigned very small probability early on in the search routine. The annealer was therefore unable to escape the second local minimum it found, and eventually terminated based on the minimum solution acceptance criterion; the acceptance ratio for the 36^{th} Markov chain dropped below χ_f.

The effect of varying L_k for both cooling schemes was explored by performing 50 runs of the SA algorithm on the 5D-RF using different random number sequences. This was achieved by specifying 50 different *seeds* to the random number generator. Figure 5 plots the value of the best objective found averaged over the 50 runs. The uncertainty bars correspond to the mean plus and minus

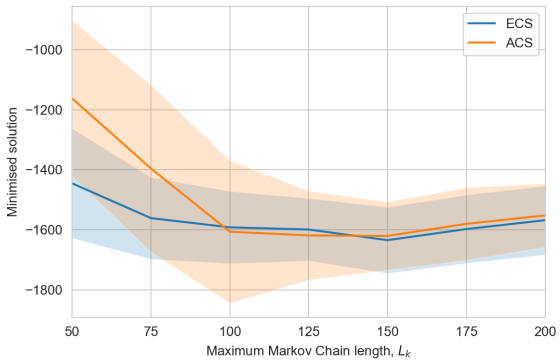


Figure 5: Variation of the minimised 5D-RF for the exponential and adaptive cooling schedules with L_k averaged over 50 runs. The shaded area represents the mean plus and minus the standard deviation.

the standard deviation in the value of the best objective found. For the ECS, their width remains fairly constant, highlighting that the *consistency* of the annealing schedule is similar over the range of L_k's tested. The width of the uncertainty bars for the ACS is large for lower values of L_k; the high standard deviation indicates that some runs might yield potentially very good performance, but others quite poor performance. For both the exponential and adaptive cooling schemes, the lowest average best solution occurred for a maximum chain length of $L_k = 150$. With a lower overall variance and fractionally lower mean minimised solution, the exponential cooling scheme with $L_k = 150$ was selected as the optimal annealing schedule for the minimisation of the 5D-RF.

3.3 Generation of Trial Solutions

The next set of parameters investigated were those relating to the diagonal matrix D, which scales the elements of the vector of random variables $\mathbf{u}$ in the generation of trial solutions, as given by eq. (4). These parameters are of particular importance, as they govern the distribution of obtainable solutions.

3.3.1 Matrix Upper Limit

Figure 6 plots the value of the best objective found averaged over 50 runs as a function of the upper limit D_u imposed on the matrix D. The average best solution is minimised for a value $D_u = 50$. Lower values of D_u restrict the magnitude of the maximum permissible change in each variable of $\mathbf{x}_i$. The mobility of the annealer is inhibited, and it struggles to explore the entirety of the search-space. Therefore, it focuses itself on local minima encountered early on in the optimisation process, and often fails to reach the global minima. Conversely, higher values of D_u allow for larger step sizes. Individual values of D_u can grow larger, and the SA algorithm is able to traverse a greater portion of the search-space, even at low temperatures. As a result, its ability to refine the search to regions where the objective function takes lower values is hampered, resulting in larger average minimised objective function values.

3.3.2 Damping Constant

From eq. (5), the damping constant α determines the rate at which information from the matrix R is folded into the matrix D. The variation of the best objective found with α, averaged over 50 runs, is plot in Figure 7. The average best objective is minimised for a value $\alpha = 0.1$. Higher values of α incorporate information regarding the magnitude of the successful changes at a faster rate, resulting in the elements of D being increased more quickly. Since the elements of D specify

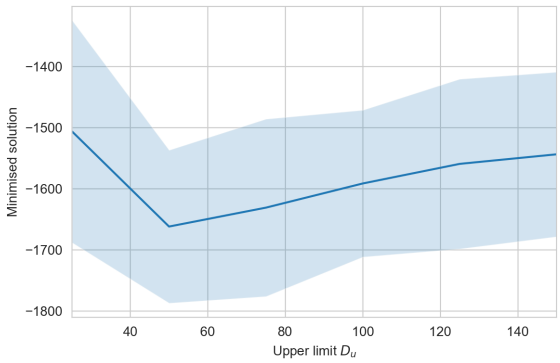


Figure 6: Minimised 5D-RF as a function of the upper limit D_u, averaged over 50 runs. The shaded area represents the mean plus and minus the standard deviation.

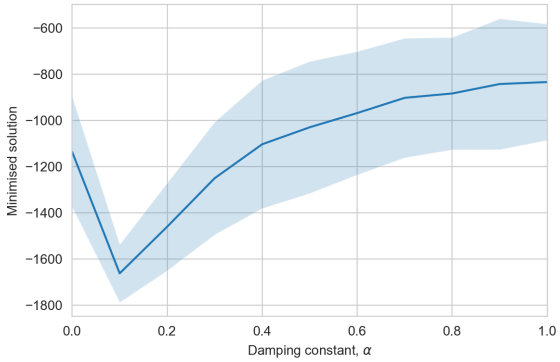


Figure 7: Variation of the value of the minimised 5D-RF as a function of the damping constant α, averaged over 50 runs. The shaded area represents the mean plus and minus the standard deviation.

the maximum change allowed in each variable, larger elements of D allow for larger subsequent step sizes and hence greater exploration of the search-space. The converse is true for smaller α, in which the elements of D are updated at a slower rate, leading to greater exploitation of good solutions. The optimal parameter setting $\alpha = 0.1$ provides a trade-off between these two traits, a condition required for an effective search.

3.3.3 Weighting Constant

Closely tied to the damping constant is the weighting constant ω, which from eq. (5) can be seen to control the weighting with which information from R is folded into D. Figure 8 plots the best objective found as ω was varied, with α fixed to its optimal value of 0.1, averaged over 50 runs. The average best objective wass minimised for a value $\omega = 2.1$. This pair of constants $(\alpha, \omega) = (0.1, 2.1)$ is consistent with the findings made by Parks (1990): this configuration tunes the maximum step size towards a value which yields acceptable changes as the search progresses.

3.4 Final Configuration

To evaluate the effect made by setting the investigated parameters to their optimal values, the minimised solution of the 5D-RF was recorded as a function of SA iterations for both the default configuration and the final, optimised configuration. Figure 9 plots the minimised solution averaged over 50 runs for both configurations. There was a significant performance gain obtained in the value of the minimised 5D-RF after 2500 iterations: on average, the optimal configuration returned

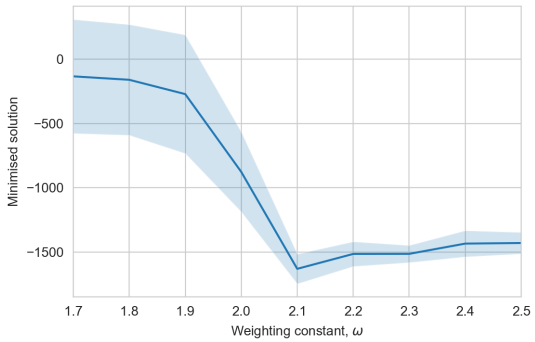


Figure 8: Variation of the value of the minimised 5D-RF as a function of the weighting constant ω, averaged over 50 runs. The shaded area represents the mean plus and minus the standard deviation.

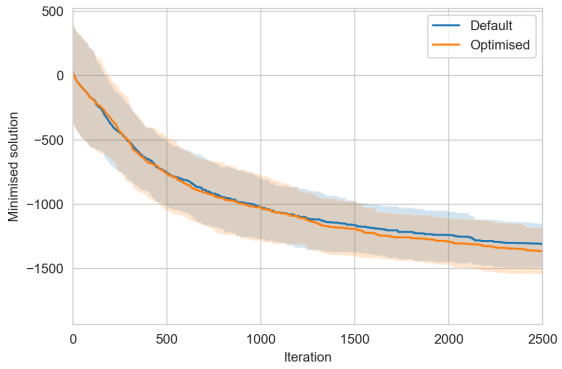


Figure 9: Evolution of the value of the minimised 5D-RF with SA iterations for both the default and optimal parameter configurations, averaged over 50 runs. The shaded area represents the mean plus and minus the standard deviation.

a minimised solution 5.88% lower than that of the default configuration. One potential reason why the difference was not greater is that many of the default solution generation parameters were set close to or equal to their optimal settings; only the cooling scheme, maximum Markov chain length L_k and matrix upper limit D_u parameters were modified. The uncertainty in the minimised solution for the optimal configuration, represented by the shaded area of the orange plot in Figure 9, was on average 5.40% larger than that of the default configuration, indicating that the optimal configuration performs less consistently than the default. In the majority of optimisation situations, the improvement in mean minimised solution for the 5D-RF is likely to outweigh the small increase in the standard deviation of the solutions, making the optimal configuration the better choice of parameter setting.

4 Evolution Strategies

4.1 Outline of Implementation

A Python implementation of an Evolution Strategy (ES), based primarily on the recommendations presented by Schwefel (1981), was devised, the code for which is presented in Appendix A.2. Where possible, explicit loops were avoided, and the code written using `NumPy` array expressions. By *vectorising* the code in this way, operations are applied simultaneously over many entries, in contrast to one entry at a time (as for a loop). Such parallelism yields significant cpu time performance gains:

vectorised array operations run one to two orders of magnitude faster than their pure Python loop counterparts (McKinney 2017).

Unlike Schwefel's original implementation, in which an *initial population* is generated by seeding an individual at a random point in the search-space and repeatedly applying *mutations*, an initial population is generated according to the method outlined by Bäck (1996): population members are conceived by sampling from an n-dimensional uniform distribution over the feasible region. The initial *covariance matrix* is generated by setting the diagonal variance elements $\sigma_i^2(0) = 9.0$, and the off-diagonal cross-covariance terms $c_{ij}(0) = 0$. Selecting small initial variances $\sigma_i^2(0)$ prevents the algorithm diverging when the number of parents μ is large (Bäck 1996). The operations in the algorithm work primarily in terms of *rotation angles* α_{ij}, which are related to the covariances by:

$$\alpha_{ij} = \frac{1}{2} \tan^{-1} \left(\frac{2c_{ij}}{\sigma_i^2 - \sigma_j^2} \right). \tag{12}$$

In doing so, it is easier to guarantee the covariance matrix remains *positive semi-definite* (PSD).

For (μ, λ)-selection, the μ best individuals are selected *deterministically* from the pool of λ offspring. Whereas in $(\mu+\lambda)$-selection, the μ best individuals are selected from the combined pool of μ previous parents and their λ offspring. A new generation of λ offspring are created by first mutating the standard deviations and rotation angles (collectively referred to as the *strategy parameters*):

$$\sigma_i' = \sigma_i \exp\left(\tau' \times \aleph_0 + \tau \times \aleph_i\right) \tag{13}$$

$$\alpha_{ij}' = \alpha_{ij} + \beta \times \aleph_{ij} \tag{14}$$

where the $\aleph$s are realisations of normally distributed one dimensional random variables $\mathcal{N}(0, 1)$ with mean zero and standard deviation one, and τ, τ' and β are algorithm *control parameters* with values as suggested by Schwefel (1977):

$$\tau = \frac{1}{\sqrt{2\sqrt{n}}}; \quad \tau' = \frac{1}{\sqrt{2n}}; \quad \beta = \frac{5\pi}{180}.$$

The control variables are then mutated by adding a perturbation $\mathbf{n}$:

$$\mathbf{x}' = \mathbf{x} + \mathbf{n}, \tag{15}$$

where the vector $\mathbf{n}$ is drawn from an n-dimensional Gaussian distribution $\mathcal{N}(\mathbf{0}, C^{-1})$ with all-zero expectation vector and covariance matrix $C^{-1} = \{c_{ij}'\}$, the elements of which are given by:

$$c_{ij}' = \frac{1}{2} \left(\sigma_i'^2 - \sigma_j'^2 \right) \tan\left(2\alpha_{ij}'\right). \tag{16}$$

A small positive constant ϵ_c is added to the diagonal elements of C^{-1}. Provided ϵ_c is chosen large enough, this modified matrix is guaranteed to PSD, and hence a valid covariance matrix.

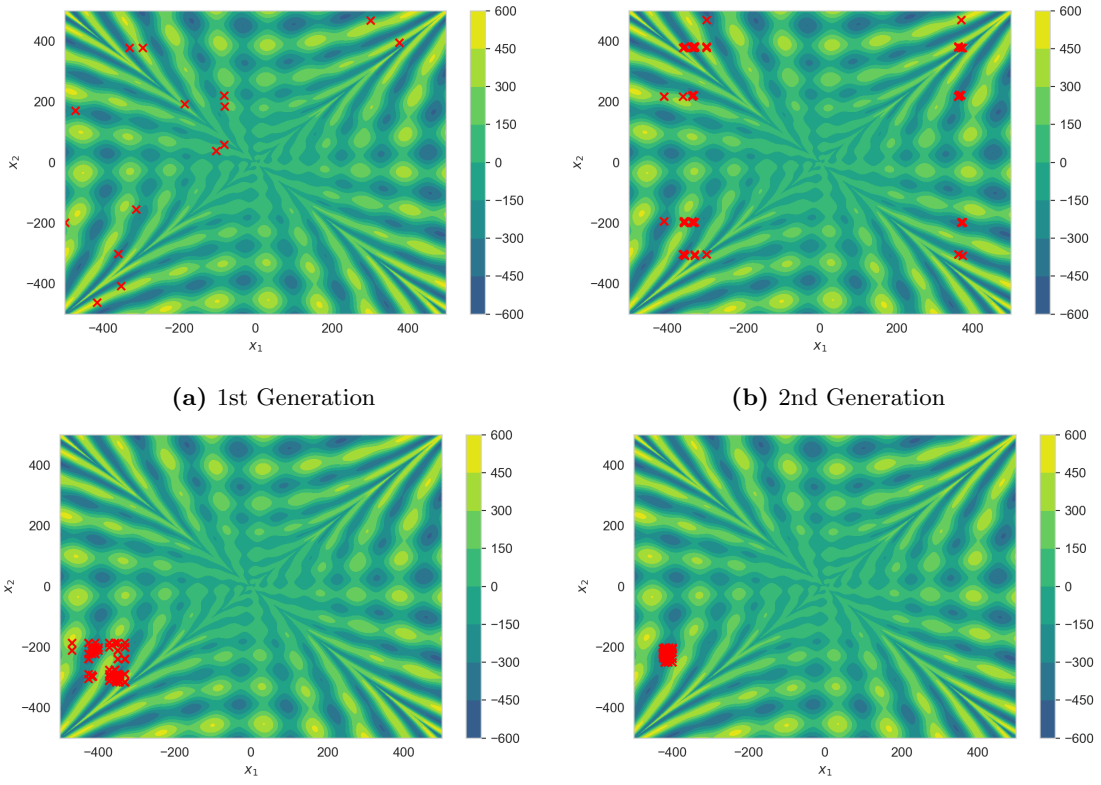
The multiplicative mutation process can result in individual standard deviations becoming too small. To prevent this, the algorithm constrains all standard deviation to a lower bound ϵ_σ:

$$\sigma_i' < \varepsilon_\sigma \Rightarrow \sigma_i := \varepsilon_\sigma. \tag{17}$$

Rotation angles are kept in the feasible range of angle values $[-\pi, \pi]$ by circularly mapping angles α_{ij}' which exceed the limits:

$$\left|\alpha_{ij}'\right| > \pi \Rightarrow \alpha_{ij}' := \alpha_{ij}' - 2\pi \cdot \text{sign}\left(\alpha_{ij}'\right). \tag{18}$$

The μ mutated control variables are *recombined* through *discrete recombination* to produce λ offspring solution. The component of each offspring solution is copied with probability 0.5 from each of its two randomly parent solutions. The strategy parameters are recombined using *intermediate*



(a) 1st Generation

(b) 2nd Generation

(c) 4th Generation

(d) 6th Generation

Figure 10: Evolution of the population distribution for the minimisation of the 2D-RF using the ES algorithm with the default parameter configuration.

recombination: solution parameters are formed as a weighted average of the components from two randomly selected parents.

The inequality constraints of the problem are handled as in Schwefel's original algorithm simply by removing offspring which violate the constraints, and repeating the creation of new solutions (through mutation and recombination) until the required number of offspring λ is obtained.

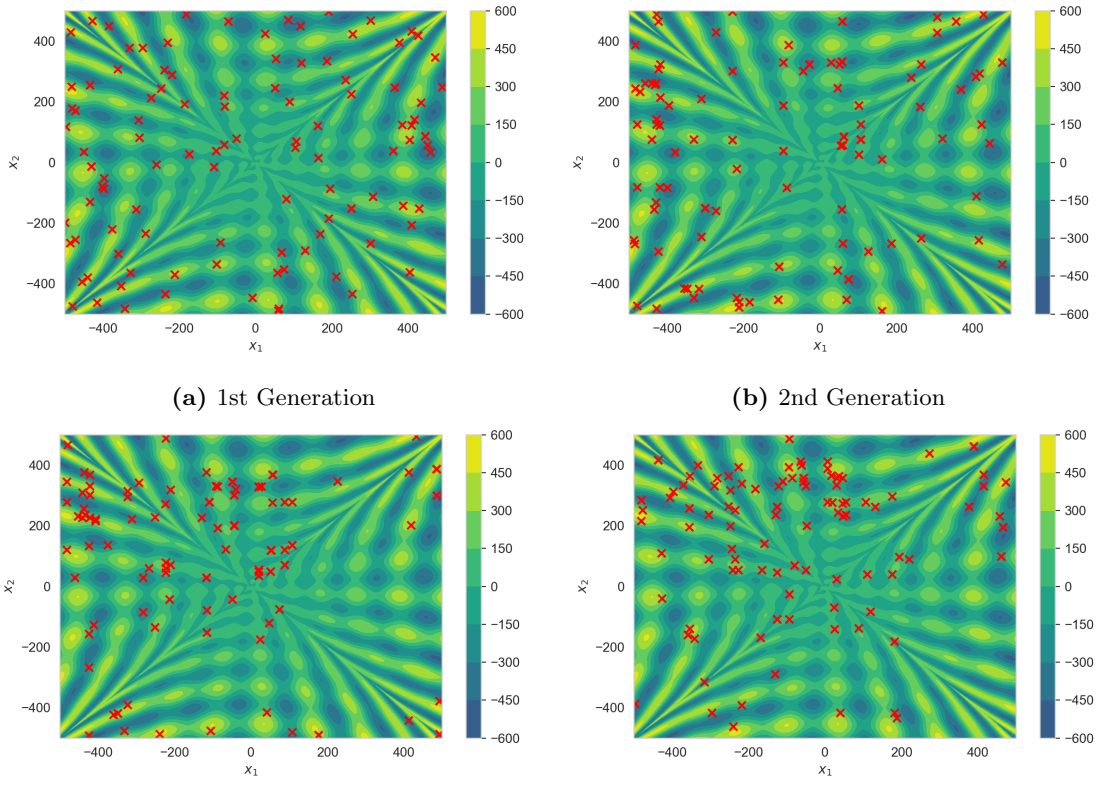
The search is terminated when the absolute difference between the worst and best solutions within a parent population falls below a user-specified threshold $\epsilon > 0$:

$$|f_w - f_b| < \epsilon \tag{19}$$

where $f_w = \max_{\mathbf{x} \in \{\mathbf{x}_1 \ldots \mathbf{x}_\mu\}} f(\mathbf{x})$ and $f_b = \min_{\mathbf{x} \in \{\mathbf{x}_1 \ldots \mathbf{x}_\mu\}} f(\mathbf{x})$, or if the number of objective function evaluations exceeds the prescribed limit of 10,000.

4.2 2-Dimensional Rana Function

Figure 10 plots the population distribution obtained for a single run of the ES algorithm applied to the 2D-RF. The default parameter settings, as given in Appendix A.2, were used for this run. Each member of the 1st, 2nd, 4th and 6th generations is shown by a red cross. The variance in the spatial distribution of the population reduces as the algorithm progresses, converging to a local optimum at (-418, -220). A series of other runs are show in Appendix B. Together, they demonstrate the optimisation method worked as anticipated.



(a) 1st Generation

(b) 2nd Generation

(c) 4th Generation

(d) 6th Generation

Figure 11: Evolution of the population distribution for the minimisation of the 2D-RF using the ES algorithm for (μ, λ)-selection with the extreme case of $\mu = \lambda = 15$.

4.3 Selection

The selection operator provides the means by which ESs direct their search towards promising regions of the search space. At any given generation, selection exploits information represented within the population to reduce diversity by focusing on individual solutions where the evaluated objective function is lower.

There are two candidate selection schemes available for ESs: either (μ, λ), in which the μ best individuals are selected from the λ offspring, or $(\mu + \lambda)$, in which the μ best parents are selected from the union of the μ previous parents and their λ offspring.

Each potential candidate schemes were trialled for different ratios of $\mu : \lambda$. To ensure (μ, λ)-selection remained deterministic, the number of parents was kept lower than the number of offspring. In doing so, the random walk behaviour that occurs in the extreme case when $\mu = \lambda$ was avoided. Figure 11 demonstrates this random walk behaviour for the 2-dimensional case; it plots snapshots of the population distribution for the minimisation of the 2D-RF using the same algorithm configurations as Figure 10, but with one change that $\mu = \lambda$ instead of $\mu = \lambda/7$.

The performance of these potential schemes was investigated by applying the ES algorithm to the 5D-RF and recording the minimised solution. The number of parents was fixed as $\mu = 15$, and the ratio λ/μ varied in integer intervals from $\lambda/\mu = 5$ to $\lambda/\mu = 9$ The value of the best objective found for the 5D-RF averaged over 50 runs is plot as a function of λ/μ for both selection schemes in Figure 12. It shows that on average, the minimised solution obtained by the $(\mu + \lambda)$ scheme is lower than that from the (μ, λ)-selection over the range of ratios tested. Furthermore, the width of the error bars, corresponding to the mean plus and minus the standard deviation, remain tighter for $(\mu + \lambda)$-selection compared to (μ, λ). Thus for the 5D-RF, a $(\mu + \lambda)$ selection scheme on average

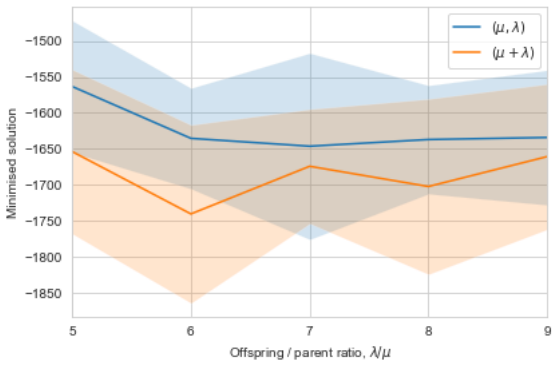


Figure 12: Value of the minimised 5D-RF as a function of the offspring to parent ratio, averaged over 50 runs. The shaded area represents the mean plus and minus the standard deviation.

yields a lower minimised solution with greater consistency.

One potential reason for the superiority of $(\mu+\lambda)$-selection for the 5D-RF is the fact that it guarantees that the best individuals from the combined parent-offspring population survive to become the subsequent parents. Consequently, it produces a monotonic course of evolution, remembering good solutions from the previous reproduction cycle and carrying them forward to the next cycle. This is in contrast to (μ, λ)-selection, in which individuals may only be selected in one reproduction cycle.

The best average solution found for the (μ, λ) selection scheme occurred for a ratio of $\mu : \lambda$ of $1 : 7$. This is consistent with the findings of Schwefel (1987). For $(\mu + \lambda)$-selection, a lower ratio of $1 : 6$ was found to be optimal, returning on average the best overall solution for the two selection schemes trialled.

The ratio $\mu : \lambda$ controls the degree of exploitation in the search, often termed the *selection pressure* (Bäck 1994). Higher selection pressures place greater emphasis on better solutions, leading to increased exploitation. On the other hand, lower selection pressure enables worse solutions to survive, allowing for greater exploration. The optimal selection pressures provide a good balance between exploitation and exploration, a condition critical in order to achieve reasonable behaviour of an ES in the case of a complicated optimisation problem, such as the 5-dimensional Rana Function. Furthermore, selection pressures in this range are low enough to avoid *punctuated equilibrium* evolution (Gould & Eldredge 1977, Schwefel 1995), an undesirable evolutionary phenomenon in which there are isolated episodes of rapid speciation amidst long periods of little or no change.

4.4 Mutation

Mutation is the primary source of genetic variation in an ES algorithm. It serves the purpose of introducing *innovations*, a means by which the optimisation method explores the search-space (Bäck 1996, Schwefel 1995).

From eq. (13), the control parameters τ and τ' dictate the standard deviations ('step sizes') of the $\mathcal{N}(0, 1)$ normally distributed random variables $\aleph_i$ and $\aleph_0$. These random variables are exponentiated, and then mutliplied with σ_i to produce changes in the standard deviations. Similarly, from eq. (14), the parameter β governs the standard deviation of the perturbation applied to α_{ij}.

Under the recommendation of Schwefel (1977), by default the parameters take the following values when the ES algorithm is applied to the 5D-RF:

$$\tau = 0.473; \quad \tau' = 0.316; \quad \beta = 0.0873.$$

The effect of modifying the control parameters from their recommended values was investigated first by fixing $\beta = 0.0873$ and jointly varying τ and τ'. The parameters τ and τ' were then fixed to their optimal values, and β varied.

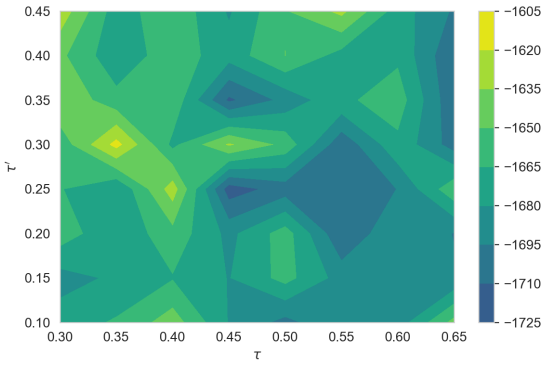


Figure 13: Minimised value of the 5D-RF plot as a function of the control parameters τ and τ', averaged over 50 runs. The parameter β was fixed to 0.0873.

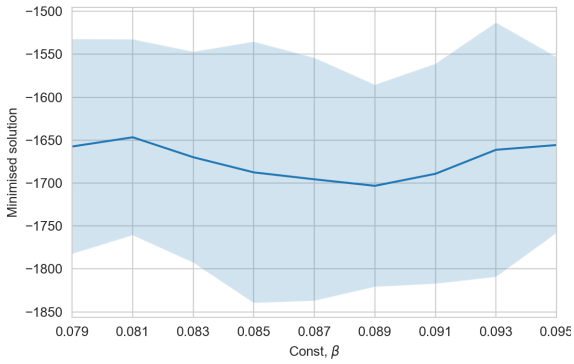


Figure 14: Minimised value of the 5D-RF plot as a function of the control parameter β, averaged over 50 runs. The parameter pair (τ, τ') were fixed to their optimal values of $(0.45, 0.25)$.

Figure 13 plots the contours of the minimised solution obtained when the ES was applied to the 5D-RF, averaged over 50 runs, as a function of τ and τ' with β held fixed. The lowest average solution occurred for a parameter pair $(\tau, \tau') = (0.45, 0.25)$. These two parameters were then fixed to their optimal values, and the minimised solution recorded as a function of β, a plot of which is shown in Figure 14. It shows that on average, the best minimised solution was obtained for $\beta = 0.089$, giving the trio of optimised control parameters:

$$\tau = 0.45; \quad \tau' = 0.25; \quad \beta = 0.089.$$

The first and last of these parameters take values close to those recommended by Schwefel (1977). However, the second parameter is significantly smaller, indicating a lower step size of individual-specific standard deviation mutations is more optimal for the ES applied to the 5D-RF.

4.5 Final Configuration

To quantify the effect of optimising the selection scheme and control parameters to the 5D-RF, the objective function was minimised using both the default and tuned ES algorithms, and the minimised solution recorded as a function of the ES generation. Figure 15 plots the evolution of the minimised solution with ES generation, averaged over 50 runs. On average, the final minimised solution for the optimised configuration was 5.43% lower than that of the default configuration, a significant performance gain. However, the uncertainty in the solution for the optimised settings was also marginally larger; the standard deviation for the tuned configuration was 3.32% greater its default counterpart. Together, these findings indicate that whilst on average the optimised parameter

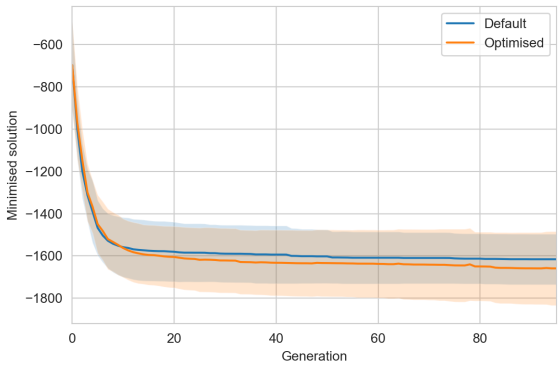


Figure 15: Variation of the minimised solution for the ES algorithm applied to the 5D-RF averaged over 50 runs for both the default and optimised parameter configurations. The shaded area represents the mean plus and minus the standard deviation.

configuration yields better minimised objective values, it does so slightly less consistently. For most applications, the improvement in the mean minimised solution is likely to outweigh the small penalty in consistency, making the optimised configuration the better choice for algorithmic performance.

5 Conclusion

Whilst convex optimisation methods work efficiently and reliable when applied to convex problems, many real-world optimisation problems violate the constraints of strict convexity. Furthermore, deterministic gradient-based methods are inapplicable for problems involving multiple minima. For more complex problems, a different class of stochastic based optimisation methods are required.

The n-dimensional Rana-Function is one such example of a multimodal function. Constraining the feasible region to the domain $x_i \in [-500, 500]$ for $i = 1, \dots n$, this function was minimised using two different heuristic-based approaches, which were individually coded in Python. Critical to the performance of these methods is problem specific tuning of algorithmic parameters. A systematic investigation into a sub-set of parameters was performed for each of the methods presented.

The first algorithm explored was simulated annealing, for which there was shown to be a close tie-in between the cooling scheme and maximum Markov chain length. The exponential cooling scheme provided consistently lower minimised solutions compared to the adaptive scheme when applied to the 5D-RF. Modifying the parameters relating to the perturbation scaling matrix D effected the degree to which the annealer could traverse the search space. Tuning the damping and weighting parameters correctly was essential for providing maximum step sizes which yielded acceptable solution changes as the search progressed. The final optimised SA configuration produced a lower mean minimised solution than its default counterpart, with only a marginal decrease in algorithmic consistency.

The second search method focused on an evolution strategy implementation. A $(\mu + \lambda)$ selection scheme produced better optimisation performance than (μ, λ)-selection. The ratio of $\mu : \lambda$ was set to provide a suitable level of selection pressure. The mutation control parameters were varied in a sequential manner, and optimal values found to give appropriate step sizes for mutation of the strategy parameters. On average, the final optimised configuration produced a lower minimised solution, with only a slight increase in the standard deviation of the best solution.

References

Aarts, E., Korst, J. & Laarhoven, van, P. (1997), *Simulated Annealing*, Wiley-Interscience.

Bäck, T. (1994), Selective pressure in evolutionary algorithms: A characterization of selection mechanisms, *in* 'Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence', Vol. 1, pp. 57–62.

Bäck, T. (1996), *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*, Oxford University Press, Inc., USA.

Boyd, S. & Vandenberghe, L. (2004), *Convex Optimization*, Cambridge University Press.

Darwin, C. (1859), *On the Origin of Species by Means of Natural Selection*, Murray, London.

Gould, S. J. & Eldredge, N. (1977), 'Punctuated Equilibria: The Tempo and Mode of Evolution Reconsidered', *Paleobiology* **3**(2), 115–151.

Granville, V., Krivanek, M. & Rasson, J. . (1994), 'Simulated annealing: a proof of convergence', *IEEE Transactions on Pattern Analysis and Machine Intelligence* **16**, 652–656.

Huang, M. D., Romeo, F. & Sangiovanni-Vincentelli (1986), 'An Efficient General Cooling Schedule for Simulated Annealing', *Proceedings of the IEEE International conference on Computer Aided Design* pp. 381–384.

Kirkpatrick, S. (1984), 'Optimization by Simulated Annealing: Quantitative Studies', *Journal of Statistical Physics* **34**, 975–986.

Kirkpatrick, S., Gelatt, C. D. & Vecchi, M. P. (1982), 'Optimization by Simulated Annealing', *IBM Research Report* **RC 9355**.

McKinney, W. (2017), *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*, 2nd edn, O'Reilly Media, Inc.

Nesterov, Y. & Nemirovskii, A. (1994), *Interior Point Polynomial Algorithms in Convex Programming*, Society for Industrial and Applied Mathematics, Philadelphia, Pa.

Parks, G. T. (1990), 'An Intelligent Stochastic Optimization Routine for Nuclear Fuel Cycle Design', *Nuclear Technology* **89**, 233–246.

Schneider, J. J. & Kirkpatrick., S. (2006), *Stochastic optimization*, Scientific computation, Springer, Berlin.

Schwefel, H. (1987), Collective Phenomena in Evolutionary Systems, *in* 'Preprints of the 31st Annual Meeting', International Society for General System Research, Budapest, pp. 1025–1033.

Schwefel, H.-P. (1977), *Numerische Optimierung von Computermodellen mittels der Evolutionsstrategie*, Vol. 26, Birkhäuser, Basel Stuttgart.

Schwefel, H.-P. (1981), *Numerical Optimization of Computer Models*, John Wiley & Sons, Inc., USA.

Schwefel, H.-P. (1995), *Evolution and Optimum Seeking*, John Wiley & Sons, Inc., USA.

Taylor, C. R. (1993), *Applications of dynamic programming to agricultural decision problems*, Westview Press, Boulder.

Whitley, D., Rana, S., Dzubera, J. & Mathias, K. E. (1996), 'Evaluating Evolutionary Algorithms', *Artificial Intelligence* **85**, 245 – 276.

Whitley, L. D., Mathias, K. E., Rana, S. B. & Dzubera, J. (1995), Building Better Test Functions, *in* 'Proceedings of the 6th International Conference on Genetic Algorithms', Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, p. 239–247.

Appendix

A Code Listings

A.1 Simulated Annealing

```
import numpy as np
from numpy.linalg import norm


def f(x):
    """
    Function which evaluates the Rana-Function for an n-dimensional solution.

    Parameters
    ----------
    x : (n, ) array
        Solution to be evaluated.

    Returns
    -------
    f_eval : float
        Objective function evaluated at the input x.
    """
    # Initialise the output value
    f_eval = 0
    # Summation over (n-1) dimensions
    for i in range(x.shape[0] - 1):
        f_eval += x[i] * np.cos(np.sqrt(np.abs(x[i + 1] + x[i] + 1))) * np.sin(
            np.sqrt(np.abs(x[i + 1] - x[i] + 1))) + (1 + x[i + 1]) * np.cos(
            np.sqrt(np.abs(x[i + 1] - x[i] + 1))) * np.sin(np.sqrt(np.abs(x[i + 1]
                    + x[i] + 1)))
    return f_eval


class SA:
    """
    Simulated annealing optimisation to find the minimum of an objective function
        f. The initial annealing temperature is found by performing an initial
        search and selecting a temperature which yields a specified average
        probability of acceptance for increases in f. The trial solution formula
        suggested by Parks [1990] takes the form:
    x_{i+1} = x_{i} + Du,
```

```
33      where u ~ U(-1,1), and D is a diagonal matrix which specifies the maximum
            change permitted in each variable. D is updated after successful changes
            to the objective function by folding in information regarding the
            magnitudes of successful changes made. The temperature is decremented
            adaptively by the formulation from Huang et al. [1986] once a given number
            of trials have been performed at the current temperature, or if a minimum
            number of acceptances have been made, whichever comes first. If a given
            number of iterations have elapsed without improvement to the current best
            solution, a restart is performed, and the search resumed from the current
            best solution. The archive stores the current best solutions, based on a
            Euclidean distance dissimilarity criterion.
34      """
35
36      def __init__(self, f, n, bound, max_iter=10000, seed=0, max_chain=100,
                chi_0=0.8, alpha=0.1, omega=2.1,
37                   d_u=None, d_l=None, cooling_scheme='adaptive', max_restart=2000,
                             chi_f=0.01, archive_size=25,
38                   d_min=None, d_sim=None):
39          """
40          Class attribute initialisation.
41
42          Parameters
43          ----------
44          f : func
45              Objective function to be minimised.
46          n : int
47              Number of dimensions over which to optimise.
48          bound : float
49              Symmetric bound (+/-) on the domain of the control variables.
50          max_iter : int, optional
51              Maximum number of objective function evaluations. The default is
            10000.
52          seed : int, optional
53              Random number generator seed for reproducibility. The default is 0.
54          max_chain : int, optional
55              Number of iterations after which to terminate Markov chain and
            decrement temperature. The default is 100.
56          chi_0 : float, optional
57              Avg probability of accepting increases in f during initial search. The
            default is 0.8.
58          alpha : float, optional
59              Damping constant. The default is 0.1.
60          omega : float, optional
61              Weighting constant. The default is 2.1.
62          d_u : float, optional
63              Upper limit on the elements of the diagonal matrix D. If d_u is None
            (default), d_u is set to bound / 5.
64          d_l : float, optional
65              Lower limit on the elements of the diagonal matrix D. If d_l is None
            (default), d_l is set to d_u / 1e5.
66          cooling_scheme : str, optional
67              Choice of cooling scheme, either 'exponential' (Kirkpatrick et al.
            [1982]) or 'adaptive' (Huang et al. [1986]). The default is 'adaptive'.
68          max_restart : int, optional
```

```
69              Number of iterations after which to restart search from the current
            best solution. The default is 2000.
70          chi_f : float, optional
71              Solution acceptance ratio termination threshold. The default is 0.01.
72          archive_size : int, optional
73              Number of solutions to store in the archive. The default is 25.
74          d_min : float, optional
75              Archive dissimilarity threshold. If d_min is None (default), d_min is
            set to bound / 100.
76          d_sim : float, optional
77              Archive dissimilarity threshold. If d_sim is None (default), d_sim is
            set to d_min / 100.
78          """
79          # assertion check
80          assert cooling_scheme in ['exponential', 'adaptive'], \
81              "cooling scheme must either be 'exponential' or 'adaptive'"
82          # initialise attributes
83          self.f = f
84          self.n = n
85          self.bound = np.abs(bound)
86          self.max_iter = max_iter
87          self.seed = seed
88          self.max_chain = max_chain
89          self.chi_0 = chi_0
90          self.alpha = alpha
91          self.omega = omega
92          if d_u:
93              self.d_u = d_u
94          else:
95              self.d_u = self.bound / 5
96          if d_l:
97              self.d_l = d_l
98          else:
99              self.d_l = self.d_u / 1e5
100         self.cooling_scheme = cooling_scheme
101         self.max_restart = max_restart
102         self.chi_f = chi_f
103         self.archive_size = archive_size
104         if d_min:
105             self.d_min = d_min
106         else:
107             self.d_min = self.bound / 100
108         if d_sim:
109             self.d_sim = d_sim
110         else:
111             self.d_sim = self.d_min / 100
112         # random seed for reproducibility
113         np.random.seed(seed=seed)
114         # generate initial solution by drawing from a uniform random variable over
                the feasible region
115         self.x0 = np.random.uniform(-bound, bound, n)
116         self.f_x0 = f(self.x0)
117         # initialise archive
118         self.archive_x = [self.x0]
119         self.archive_f = [self.f_x0]
```

```
120         # generate non-zero elements of the diagonal matrix D, represented as a
                (n, ) vector d
121         self.d = np.ones(n)
122         # initialise iteration and acceptance counters
123         self.iter = 1
124         self.acc = 1
125 
126     def initialise_T(self):
127         """
128         Method which initialises the search temperature T_0 by performing an
            initial search in which all increases in f are accepted. T_0 is calculated
            to give an average probability chi_0 of a solution that increases f.
129 
130         Returns
131         ----------
132         x_prev : (n, ) array
133             Vector of current solution.
134         f_prev : float
135             Objective function evaluated at current solution.
136         T_0 : float
137             Initial search temperature.
138         """
139         # initial solution x0
140         x_prev = self.x0
141         # evaluate objective function at initial solution
142         f_prev = self.f_x0
143         # array of objective increases
144         df_plus = []
145 
146         # run the initial search for specified maximum length of Markov chain
147         while self.iter < self.max_chain:
148             # generate a uniform rv in the range (-1,1) of equal dimension as x
149             u = np.random.uniform(-1, 1, self.n)
150             # find the proposal step dx
151             dx = self.d * u
152             # perturb the current position x_prev by the proposal step dx
153             x_new = x_prev + dx
154             # reject proposal if outside of feasible region
155             if max(abs(x_new)) > self.bound:
156                 # increment iteration counter
157                 self.iter += 1
158                 continue
159             # evaluate change in objective function
160             f_new = self.f(x_new)
161             df = f_new - f_prev
162             # record increases in objective function
163             if df > 0:
164                 df_plus.append(df)
165             # update d only if objective function is decreased(?)
166             r = np.abs(dx)
167             self.d = (1 - self.alpha) * self.d + self.alpha * self.omega * r
168             self.d = np.clip(self.d, self.d_l, self.d_u)
169             # accept all feasible trial solutions
170             x_prev = x_new
171             f_prev = f_new
```

```
172                 # update archive
173                 self.update_archive(x_prev, f_prev)
174                 # increment iteration and acceptance counters
175                 self.iter += 1
176                 self.acc += 1
177 
178             # average objective function increase
179             df_bar = np.mean(df_plus)
180             T_0 = - df_bar / np.log(self.chi_0)
181             return x_prev, f_prev, T_0
182 
183         def update_T(self, T, f_T):
184             """
185             Method which updates the annealing temperature according to either: the
                exponential CS outlined by Kirkpatrick et al. [1982]; the adaptive CS
                outlined by Huang et al. [1986].
186 
187             Parameters
188             ----------
189             T : float
190                 Current temperature.
191             f_T : (k, ) array
192                 Vector of objective function acceptances at current temperature.
193 
194             Returns
195             ----------
196             alpha * T : float
197                 Decremented temperature.
198             """
199             if self.cooling_scheme == 'exponential':
200                 alpha = 0.95
201             else:
202                 if len(f_T) > 1:
203                     sigma = np.std(f_T)
204                     alpha = max(0.5, np.exp(-0.7 * T / sigma))
205                 else:
206                     alpha = 0.95
207             return alpha * T
208 
209         def update_archive(self, x_j, f_j):
210             """
211             Method which updates the class archives of best solutions.
212 
213             Parameters
214             ----------
215             x_j : (n, ) array
216                 A new n-dimensional candidate solution for archiving.
217             f_j : float
218                 Objective function evaluated at the candidate solution x_j.
219             """
220             # worst archived solution
221             g_ind = np.argmax(self.archive_f)
222             f_g = self.archive_f[g_ind]
223 
224             # Euclidean distance between new and archived solutions
```

```
225             distance = norm(self.archive_x - x_j, axis=1)
226             # archived solution x_e which most closely resembles new solution x_j
227             e_ind = np.argmin(distance)
228             d_ej = distance[e_ind]
229             f_e = self.archive_f[e_ind]
230 
231             # archive is not full
232             if len(self.archive_x) < self.archive_size:
233                 # archive new solution x_j if it is sufficiently dissimilar to all the
                        solutions archived
234                 if d_ej > self.d_min:
235                     self.archive_x.append(x_j)
236                     self.archive_f.append(f_j)
237 
238             # archive is full
239             else:
240                 # archive x_j if it is sufficiently dissimilar to all the solutions
                        archived and better than the worst
241                 if d_ej > self.d_min and f_j < f_g:
242                     # new solution x_j replaces the worst archived solution x_g
243                     self.archive_x.pop(g_ind)
244                     self.archive_f.pop(g_ind)
245                     self.archive_x.append(x_j)
246                     self.archive_f.append(f_j)
247                 # archive x_j if it is not sufficiently dissimilar to all the
                        solutions archived and better than the worst
248                 if d_ej < self.d_min and f_j < f_g:
249                     # new solution x_j replaces the closest archived solution x_e
250                     self.archive_x.pop(e_ind)
251                     self.archive_f.pop(e_ind)
252                     self.archive_x.append(x_j)
253                     self.archive_f.append(f_j)
254                 # archive x_j if it is not the best solution so far and sufficiently
                        similar to but better than x_e
255                 if d_ej < self.d_sim and f_j < f_e:
256                     self.archive_x.pop(e_ind)
257                     self.archive_f.pop(e_ind)
258                     self.archive_x.append(x_j)
259                     self.archive_f.append(f_j)
260 
261     def run(self):
262         """
263         Method which runs the SA algorithm.
264 
265         Returns
266         -------
267         x_min : (n, ) array
268             Minimised solution.
269         f_min : float
270             Objective function evaluated at the minimised solution.
271         archive_x : (k, n) array
272             Matrix of k final archived solutions. Each control variable is of
        dimension n.
273         archive_fx : (k, ) array
```

```
274             Vector of objective function values evaluated at the k final archived
            solutions.
275             """
276             # perform initial search
277             x_prev, f_prev, T = self.initialise_T()
278             # set minimum number of acceptances for each temperature
279             eta = 0.6 * self.max_chain
280             # initialise iterations and acceptances for current temperature
281             iter_T, acc_T = 0, 0
282             # initialise restart counter
283             restart_iter = 0
284             # array of objective function acceptances at current temperature
285             f_T = [f_prev]
286
287             # run the search routine for a given maximum number of iterations
288             while self.iter < self.max_iter:
289                 # generate a uniform rv in the range (-1,1) of equal dimension as x
290                 u = np.random.uniform(-1, 1, self.n)
291                 # find the proposal step dx
292                 dx = self.d * u
293                 # generate the elements of the diagonal matrix R, represented as a
                        vector r
294                 r = np.abs(dx)
295                 # compute the actual step size associated with the proposal step dx
296                 step_size = np.sqrt(np.sum(r ** 2))
297                 # perturb the current position x_prev by the proposal step dx
298                 x_new = x_prev + dx
299                 # increment iteration counters
300                 self.iter += 1
301                 iter_T += 1
302                 restart_iter += 1
303
304                 # accept proposal if inside feasible region
305                 if max(abs(x_new)) < self.bound:
306                     # evaluate objective function at proposal solution
307                     f_new = self.f(x_new)
308                     # acceptance probability
309                     p_acc = min(1, np.exp(- (f_new - f_prev) / (T * step_size)))
310                     if p_acc > np.random.uniform(0, 1):
311                         # reset restart counter if new best solution has been found
312                         if f_new < min(self.archive_f):
313                             restart_iter = 0
314                         # update archive
315                         self.update_archive(x_new, f_new)
316                         # update non-zero elements of matrix D
317                         self.d = (1 - self.alpha) * self.d + self.alpha * self.omega *
                                r
318                         self.d = np.clip(self.d, self.d_l, self.d_u)
319                         # update array of objective function acceptances at current
                                temperature
320                         f_T.append(f_new)
321                         # accept solution
322                         x_prev = x_new
323                         f_prev = f_new
324                         # increment acceptance counters
```

```
                    acc_T += 1
                    self.acc += 1

            # check conditions for termination
            # terminate if no new best solution over whole Markov chain and
                    acceptance ratio falls below threshold chi_f
            if iter_T > self.max_chain and restart_iter > self.max_chain and acc_T
                    / iter_T < self.chi_f:
                break

            # check conditions for temperature cooling
            # decrement temperature if max Markov chain length is attained or
                    number acceptances exceeds eta
            if iter_T > self.max_chain or acc_T > eta:
                # standard deviation of objective function values at current
                        temperature
                T = self.update_T(T, f_T)
                # reset temperature specific counters and array
                iter_T, acc_T = 0, 0
                f_T = []

            # check conditions for restart
            if restart_iter > self.max_restart:
                # archive index of best solution
                best_ind = np.argmin(self.archive_f)
                # set current solution to best solution
                x_prev = self.archive_x[best_ind]
                f_prev = self.archive_f[best_ind]
                # reset restart counter
                restart_iter = 0

        # best minimised solution
        idx = np.argmin(self.archive_f)
        x_min = self.archive_x[idx]
        f_min = self.archive_f[idx]

        return x_min, f_min, self.archive_x, self.archive_f


def main():
    # number of dimensions and bound on feasible region
    n, bound = 5, 500
    sa = SA(f, n, bound)
    x_min, f_min, archive_x, archive_f = sa.run()


if __name__ == '__main__':
    main()
```

A.2 Evolution Strategies

```
import numpy as np
from numpy.linalg import norm
```

```
3
4
5   def f(x):
6       """
7       Function which evaluates the Rana-Function for an array of n-dimensional
            solutions.
8
9       Parameters
10      ----------
11      x : (k, n) array
12          Input array of k control variables, each of dimension n.
13
14      Returns
15      -------
16      f_eval : (k, ) array
17          Output array of k objective function values.
18      """
19      # Initialise the array of output values
20      f_eval = np.zeros(x.shape[0])
21      # Summation over (n-1) dimensions
22      for i in range(x.shape[1] - 1):
23          f_eval += x[:, i] * np.cos(np.sqrt(np.abs(x[:, i + 1] + x[:, i] + 1))) *
                    np.sin(
24              np.sqrt(np.abs(x[:, i + 1] - x[:, i] + 1))) + (1 + x[:, i + 1]) *
                        np.cos(
25              np.sqrt(np.abs(x[:, i + 1] - x[:, i] + 1))) *
                        np.sin(np.sqrt(np.abs(x[:, i + 1] + x[:, i] + 1)))
26      return f_eval
27
28
29  class ES:
30      """
31      Class for Evolutionary Strategy (ES) optimisation to find the minimum of an
            objective function f. An initial population x is generated by sampling
            from a uniform distribution over the feasible region. Parents are
            deterministically selected, and then mutated through a stochastic process.
            The mutated parents are recombined to form offspring. The objective
            function f is then evaluated for the newly formed offspring. The
            evolutionary cycle of selection, mutation and recombination repeats. The
            algorithm is terminated when the absolute difference in the objective
            function of the best and worst members of a parent population falls within
            a user-prescribed limit.
32      """
33
34      def __init__(self, f, n, bound, max_iter=10000, seed=0, mu=15, lambd=None,
                sel=',', epsilon=1, epsilon_s=0.05,
35                   epsilon_c=1e-6, archive_size=25, d_min=None, d_sim=None,
                            tau=None, tau_d=None, beta=None):
36          """
37          Class attribute initialisation.
38
39          Attributes
40          ----------
41          f : func
42              Objective function to be minimised.
```

```
        n : int
            Number of dimensions over which to optimise.
        bound : float
            Symmetric bound (+/-) on the domain of the control variables.
        max_iter : int, optional
            Maximum number of objective function evaluations. The default is
        10000.
        seed : int, optional
            Random number generator seed for reproducibility. The default is 0.
        mu : int, optional
            Number of parents. The default is 15.
        lambd : int, optional
            Number of offspring. If lambd is None (default), the number of
        offspring is set to 7 * mu.
        sel : str, optional
            Selection scheme, either "," or "+". The default is ','
        epsilon : float, optional
            Termination threshold. The default is 1.
        epsilon_s : float, optional
            Lower limit on the standard deviation of the control variables. The
        default is 0.05
        epsilon_c : float, optional
            Positive constant by which to scale the identity matrix I added to the
        covariance matrix. Ensures the resulting matrix is PSD. The default is
        1e-6.
        tau : float, optional
            Mutation control parameter. If tau is None (default), tau is set to 1
        / sqrt(2 * sqrt(n)) (Schwefel [1977]).
        tau_d : float, optional
            Mutation control parameter. If tau is None (default), tau_d is set to
        1 / sqrt(2 * n) (Schwefel [1977]).
        beta: float, optional
            Mutation control parameter. If beta is None (default), beta is set to
        5 * pi / 180 (Schwefel [1977]).
        archive_size : int, optional
            Number of solutions to store in the archive. The default is 25.
        d_min : float, optional
            Archive dissimilarity threshold. If d_min is None (default), d_min is
        set to bound / 100.
        d_sim : float, optional
            Archive dissimilarity threshold. If d_sim is None (default), d_sim is
        set to d_min / 100.
        """
        # assertion check- selection scheme must be (mu,lambd) or (mu + lambd)
        assert sel in [',', '+'], "selection scheme must either be ',' or '+'"
        # initialise attributes
        self.f = f
        self.n = n
        self.bound = np.abs(bound)
        self.max_iter = max_iter
        self.seed = seed
        self.mu = mu
        if lambd:
            self.lambd = lambd
        else:
```

```
            self.lambd = 7 * self.mu
        self.sel = sel
        self.epsilon = epsilon
        self.epsilon_s = epsilon_s
        self.epsilon_c = epsilon_c
        self.archive_size = archive_size
        if d_min:
            self.d_min = d_min
        else:
            self.d_min = self.bound / 100
        if d_sim:
            self.d_sim = d_sim
        else:
            self.d_sim = self.d_min / 100
        if tau:
            self.tau = tau
        else:
            self.tau = 1 / np.sqrt(2 * np.sqrt(self.n))
        if tau_d:
            self.tau_d = tau_d
        else:
            self.tau_d = 1 / np.sqrt(2 * self.n)
        if beta:
            self.beta = beta
        else:
            self.beta = 5 * np.pi / 180

    def initialise(self):
        """
        Method which initialises the ES.

        Returns
        -------
        xp : (mu, n)
            Matrix of mu initial n-dimensional parent solutions.
        fp : (mu, )
            Vector of objective function values evaluated at the mu parent
        solutions.
        sigmap : (mu, n)
            Matrix of parent standard deviations.
        alphap : (mu, n)
            Matrix of parent rotation angles.
        iters : int
            Number of objective function evaluations.
        """
        # random seed for reproducibility
        np.random.seed(seed=self.seed)
        # generate the initial population by sampling from a multivariate uniform
                distribution over the feasible region
        xp = np.random.uniform(-self.bound, self.bound, [self.mu, self.n])
        # evaluate the objective function
        fxp = self.f(xp)
        # number of objective function evaluations
        iters = len(fxp)
        # initialise all standard deviations to 3
```

```
141         sigmap = 3 * np.ones((self.mu, self.n))
142         # initialise all rotation angles to 0
143         alphap = np.zeros((self.mu, self.n, self.n))
144         return xp, fxp, sigmap, alphap, iters
145 
146     def selection(self, xp, fxp, sigmap, alphap, xo, fxo, sigmao, alphao):
147         """
148         Method which performs selection. The best mu parents are selected from the
            pool of lambd offspring (, selection), or from the combined pool of
            previous mu parents and their lambd offspring (+ selection).
149 
150         Parameters
151         ----------
152         xp : (mu, n) array
153             Matrix of mu previous parent solutions, each of dimension n.
154         fxp : (mu, ) array
155             Vector of function value evaluations for the mu previous parent
            solutions.
156         sigmap : (mu, n) array
157             Matrix of standard deviations for the previous parent solutions.
158         alphap : (mu, n, n) array
159             Matrix of rotation angles for the previous parent solutions.
160         xo : (lambd, n) array
161             Matrix of n-dimensional offspring solutions.
162         fxo : (lambd, ) array
163             Vector of function value evaluations for the lambd offspring
            solutions.
164         sigmao : (lambd, n) array
165             Matrix of standard deviations for the offpsring solutions.
166         alphao : (lambd, n, n) array
167             Matrix of rotation angles for the offspring solutions.
168 
169         Returns
170         -------
171         xp : (mu, n) array
172             Matrix of mu parent solutions, each of dimension n.
173         fxp : (mu, ) array
174             Vector of function value evaluations for the mu parent solutions.
175         sigmap : (mu, n) array
176             Matrix of standard deviations for the parent solutions.
177         alphap : (mu, n, n) array
178             Matrix of rotation angles for the parent solutions.
179         """
180         if self.sel == ',':
181             # select the mu lowest solutions from the offspring only
182             idx = np.argsort(fxo)[:self.mu]
183             xp = xo[idx]
184             fxp = fxo[idx]
185             sigmap = sigmao[idx]
186             alphap = alphao[idx]
187             return xp, fxp, sigmap, alphap
188         else:
189             # append the parents and offspring solutions to form a combined pool
                    for selection
190             xc = np.append(xo, xp, axis=0)
```

```
            fxc = np.append(fxo, fxp, axis=0)
            sigmac = np.append(sigmao, sigmap, axis=0)
            alphac = np.append(alphao, alphap, axis=0)
            # select the mu lowest solutions from the combined pool
            idx = np.argsort(fxc)[:self.mu]
            xp = xc[idx]
            fxp = fxc[idx]
            sigmap = sigmac[idx]
            alphap = alphac[idx]
            return xp, fxp, sigmap, alphap

    def mutation(self, x, sigma, alpha):
        """
        Method which performs mutation of the control variables and strategy
        parameters.

        Parameters
        ----------
        x : (mu, n) array
            Matrix of mu parent solutions, each of dimension n.
        sigma : (mu, n) array
            Matrix of parent standard deviations.
        alpha : (mu, n, n)
            Matrix of parent rotation angles.

        Returns
        -------
        xm : (mu, n) array
            Matrix of mu mutated solutions, each of dimension n.
        sigmam : (mu, n) array
            Matrix of mutated standard deviations.
        alpham : (mu, n, n)
            Matrix of mutated rotation angles.
        """
        # number of parents
        num_p = x.shape[0]
        # realisation of a N(0, 1) Gaussian, used in the mutation of all sigma
            values
        chi_0 = np.random.randn()
        # construct a (mu, n, n) matrix of random numbers drawn from N(0, 1)
            Gaussian distributions
        randn = np.random.randn(num_p, self.n, self.n)
        # construct a (mu, n, n) matrix of chi's from the realisation tmp
        # the upper triangular of tmp is taken
        # the lower triangular is constructed as the transpose of the upper
            triangular of tmp (excluding the leading diagional)
        # matrix is thus symmetric in axis 1 and 2 (chi_ij = chi_ji)
        chi = np.triu(randn) + np.transpose(np.triu(randn, 1), (0, 2, 1))
        # the (mu, n) matrix of chi_i's is taken as the diagonal elements of the
            (mu, n, n) matrix chi over axis 1 and 2
        chi_i = np.diagonal(chi, axis1=1, axis2=2)
        # mutate the standard deviations
        sigmam = sigma * np.exp(self.tau * chi_i + self.tau_d * chi_0)
        # mutate the rotation angles
        alpham = alpha + self.beta * chi
```

```
241             # compute the off-diagonal elements of the (mu, n, n) mutated covariance
                    matrix
242             # the first bracketed term is a (mu, n, n) matrix of variance differences
243             # of which the ij-th term is the difference (sigmam_i^2 - sigmam_j^2)
244             # it is constructed by adding a new axis to each of the sigma matrices
245             # such that they are (mu, n, 1) and transposing one matrix over axis 1 and
                    2
246             # the second bracketed term is a (mu, n, n) matrix of two times the
                    tangent of the mutated rotation angles
247             # the two bracketed terms are multiplied element-wise to give the (mu, n,
                    n) mutated covariance matrix
248             covm = 0.5 * (sigmam[:, :, None] ** 2 - np.transpose(sigmam[:, :, None],
                    (0, 2, 1)) ** 2) * np.tan(2 * alpham)
249             # get the shape (mu, n, n) of the mutated covariance matrix
250             s0, s1, s2 = covm.shape
251             # set the diagonal elements over axis 1 and 2 of the (mu, n, n) mutated
                    covaraince matrix to the mutated variances
252             # a small term epsilon_c is added to ensure the mutated covariance matrix
                    is PSD
253             covm.reshape(s0, -1)[:, ::s2 + 1] = sigmam ** 2 + self.epsilon_c
254             # draw a (mu, n) matrix of perturbations dx from Guassian distributions
255             # dx[i] (n, ) is drawn from a N(0, covm[i]) distribution
256             dx = np.array([np.random.multivariate_normal(mean=np.zeros(self.n),
                    cov=covmi) for covmi in covm])
257             # mutate the solution x by applying the perturbation dx
258             xm = x + dx
259             return xm, sigmam, alpham
260
261         def discrete_recombination(self, xp, no):
262             """
263             Method which discretely recombines parent solutions to form a given number
                of offspring.
264
265             Parameters
266             ----------
267             xp : (mu, n) array
268                 Matrix of mu parent solutions, each of dimension n.
269             no : int
270                 Number of offspring to generate.
271
272             Returns
273             -------
274             xo : (no, n) array
275                 Matrix of no offspring solutions, each of dimension n.
276             """
277             num_p = xp.shape[0]
278             # randomly select parents 1 and 2 from the pool of potential parents
279             x1 = xp[np.random.randint(low=0, high=num_p, size=no)]
280             x2 = xp[np.random.randint(low=0, high=num_p, size=no)]
281             # construct a binary mask for parent 1
282             x1_mask = np.random.randint(low=0, high=2, size=np.array(x1.shape))
283             # parent 2's mask is complementary to parent 1's
284             x2_mask = np.logical_not(x1_mask)
285             # recombine using masks to form offspring
286             xo = x1_mask * x1 + x2_mask * x2
```

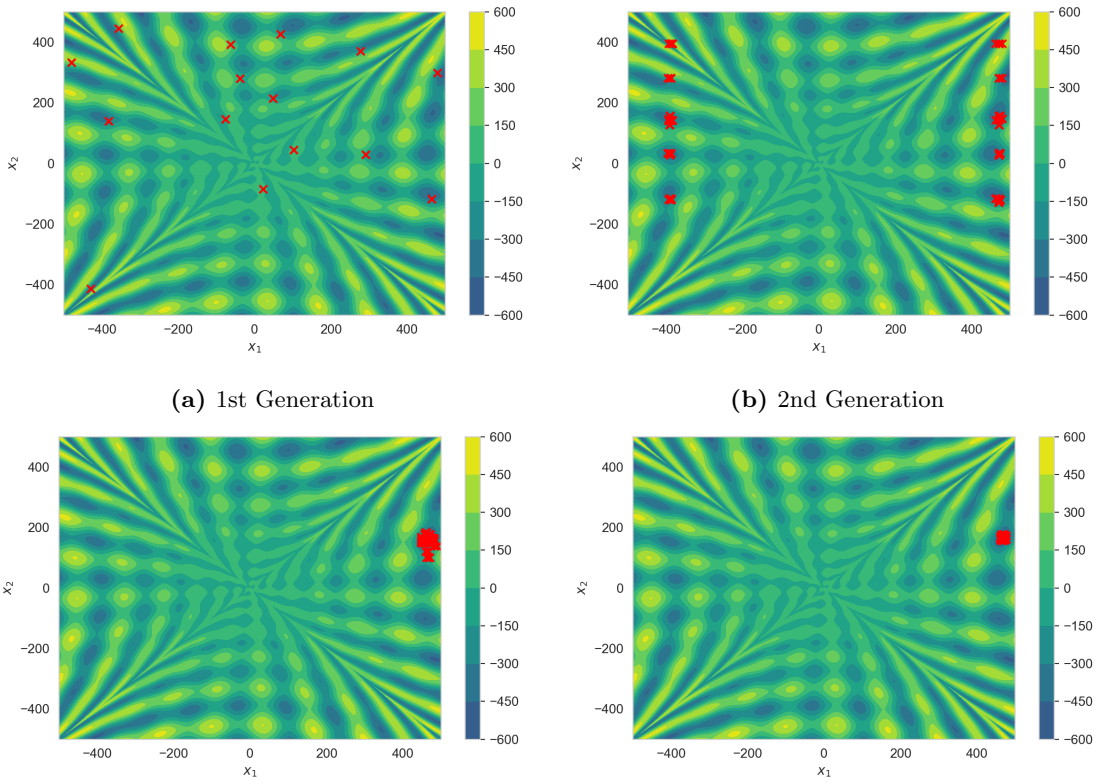
```
287             return xo
288
289         def intermediate_recombination(self, sigmap, alphap, no):
290             """
291             Method which intermediately recombines parent strategy parameters to form
                a given number of offspring strategy parameters.
292
293             Parameters
294             ----------
295             sigmap : (mu, n) array
296                 Matrix of parent standard deviations.
297             alphap : (mu, n, n) array
298                 Matrix of parent rotation angles.
299             no : int
300                 Number of offspring to generate.
301
302             Returns
303             -------
304             sigmap : (mu, n) array
305                 Matrix of offspring standard deviations.
306             alphap : (mu, n, n) array
307                 Matrix of offspring rotation angles.
308             """
309             num_p = sigmap.shape[0]
310             # randomly select parents 1 and 2 from the pool of potential parents
311             idx1 = np.random.randint(low=0, high=num_p, size=no)
312             idx2 = np.random.randint(low=0, high=num_p, size=no)
313
314             sigma1 = sigmap[idx1]
315             sigma2 = sigmap[idx2]
316             # offspring standard deviation is formed as the arithmetic mean of the
                    parent standard deviations
317             sigmao = 0.5 * (sigma1 + sigma2)
318             # force standard deviations to exceed a minimum value
319             sigmao = np.where(sigmao > self.epsilon_s, sigmao, self.epsilon_s)
320             # prevent standard deviations from exceeding a maximum value
321             sigmao = np.where(sigmao < 100, sigmao, 100)
322
323             alpha1 = alphap[idx1]
324             alpha2 = alphap[idx2]
325             # offspring rotation angles are formed as the arithmetic mean of the
                    parent rotation angles
326             alphao = 0.5 * (alpha1 + alpha2)
327             # circularly map rotation angles which lie outside the range [-pi, pi] by
                    adding/subtracting a factor of 2 * pi
328             alphao = np.where(alphao < np.pi, alphao, alphao - 2 * np.pi)
329             alphao = np.where(alphao > - np.pi, alphao, alphao + 2 * np.pi)
330             return sigmao, alphao
331
332         def update_archive(self, archive_x, archive_fx, x_j, f_j):
333             """
334             Method which updates the archive of best solutions.
335
336             Parameters
337             ----------
```

```
        archive_x : (k, n) array
            Matrix of k solutions, each of n dimensions, in the current archive.
        archive_fx : (k, ) array
            Vector of the objective function evaluated at the k solutions in the
        current archive.
        x_j : (n, ) array
            A new n-dimensional candidate solution for archiving.
        f_j : float
            Objective function evaluated at the candidate solution x_j.

        Returns
        -------
        archive_x : (l, n) array
            Matrix of k solutions, each of n dimensions, in the updated archive.
        archive_fx : (l, ) array
            Vector of the objective function evaluated at the l solutions in the
        updated archive.
        """
        # worst archived solution
        g_ind = np.argmax(archive_fx)
        f_g = archive_fx[g_ind]

        # Euclidean distance between new and archived solutions
        distance = norm(archive_x - x_j, axis=1)
        # archived solution x_e which most closely resembles new solution x_j
                (closest in Euclidean distance)
        e_ind = np.argmin(distance)
        d_ej = distance[e_ind]
        f_e = archive_fx[e_ind]

        # archive is not full
        if len(archive_x) < self.archive_size:
            # archive new solution x_j if it is sufficiently dissimilar to all the
                    solutions archived
            if d_ej > self.d_min:
                archive_x.append(x_j)
                archive_fx.append(f_j)

        # archive is full
        else:
            # archive x_j if it is sufficiently dissimilar to all the solutions
                    archived and better than the worst
            if d_ej > self.d_min and f_j < f_g:
                # new solution x_j replaces the worst archived solution x_g
                archive_x.pop(g_ind)
                archive_fx.pop(g_ind)
                archive_x.append(x_j)
                archive_fx.append(f_j)
            # archive x_j if it is not sufficiently dissimilar to all the
                    solutions archived and better than the worst
            if d_ej < self.d_min and f_j < f_g:
                # new solution x_j replaces the closest archived solution x_e
                archive_x.pop(e_ind)
                archive_fx.pop(e_ind)
                archive_x.append(x_j)
```

```
387                 archive_fx.append(f_j)
388             # archive x_j if it is not the best solution so far and sufficiently
                        similar to but better than x_e
389             if d_ej < self.d_sim and f_j < f_e:
390                 archive_x.pop(e_ind)
391                 archive_fx.pop(e_ind)
392                 archive_x.append(x_j)
393                 archive_fx.append(f_j)
394         return archive_x, archive_fx
395
396     def termination(self, fxp):
397         """
398         Method which tests the termination condition.
399
400         Parameters
401         ----------
402         fxp : (mu, ) array
403             Vector of objective function evaluations for mu parent solutions.
404
405         Returns
406         -------
407         bool
408             True if termination condition is met. False otherwise.
409         """
410         # test if absolute difference between worst and best solutions lies below
                the threshold epsilon
411         return np.abs(max(fxp) - min(fxp)) < self.epsilon
412
413     def run(self):
414         """
415         Method which runs the ES algorithm.
416
417         Returns
418         -------
419         x_min : (n, ) array
420             Minimised solution.
421         fx_min : float
422             Objective function evaluated at the minimised solution.
423         archive_x : (k, n) array
424             Matrix of k final archived solutions. Each control variable is of
        dimension n.
425         archive_fx : (k, ) array
426             Vector of objective function values evaluated at the k final archived
        solutions.
427         """
428         # generate initial solutions and control parameters
429         xp, fxp, sigmap, alphap, iters = self.initialise()
430         # form the initial archives from the initial solutions
431         archive_x = [xp[0]]
432         archive_fx = [fxp[0]]
433         for i in range(xp.shape[0]):
434             archive_x, archive_fx = self.update_archive(archive_x, archive_fx,
                    xp[i], fxp[i])
435
436         while iters < self.max_iter:
```

```
437                 if self.termination(fxp):
438                     break
439                 # initialise array of accepted offspring
440                 xo, sigmao, alphao = [], [], []
441                 # generate lambd valid offspring
442                 while len(xo) < self.lambd:
443                     no = self.lambd - len(xo)
444                     xm, sigmam, alpham = self.mutation(xp, sigmap, alphap)
445                     x_n = self.discrete_recombination(xm, no)
446                     sigma_n, alpha_n = self.intermediate_recombination(sigmam, alpham,
                            no)
447                     # keep offspring solutions which adhere to inequality constraints
448                     idx = tuple([np.max(np.abs(x_n), axis=1) < self.bound])
449                     x_n, sigma_n, alpha_n = x_n[idx], sigma_n[idx], alpha_n[idx]
450                     # append valid offspring solutions to array of accepted offspring
451                     xo = np.append(xo, x_n).reshape(len(xo) + len(x_n), self.n)
452                     sigmao = np.append(sigmao, sigma_n).reshape(len(sigmao) +
                            len(sigma_n), self.n)
453                     alphao = np.append(alphao, alpha_n).reshape(len(alphao) +
                            len(alpha_n), self.n, self.n)
454
455                 fxo = self.f(xo)
456                 iters += self.lambd
457                 xp, fxp, sigmap, alphap = self.selection(xo, fxo, sigmao, alphao, xp,
                        fxp, sigmap, alphap)
458
459                 # update archive with the parent solutions
460                 for i in range(xp.shape[0]):
461                     archive_x, archive_fx = self.update_archive(archive_x, archive_fx,
                            xp[i], fxp[i])
462
463             # best minimised solution
464             idx = np.argmin(archive_fx)
465             x_min = archive_x[idx]
466             fx_min = archive_fx[idx]
467
468             return x_min, fx_min, archive_x, archive_fx
469
470
471     def main():
472         # number of dimensions and bound on feasible region
473         n, bound = 5, 500
474         es = ES(f, n, bound)
475         x_min, fx_min, archive_x, archive_fx = es.run()
476
477
478     if __name__ == '__main__':
479         main()
```

B Evolution Strategies Population Evolution



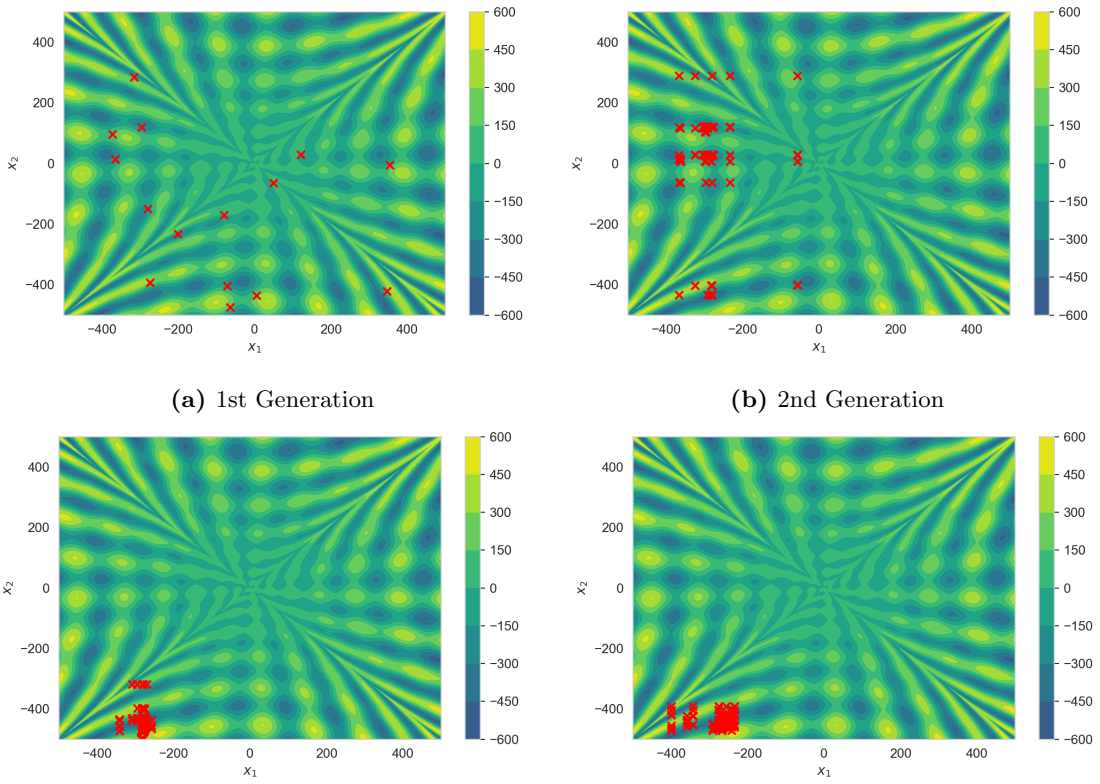
(a) 1st Generation

(b) 2nd Generation

(c) 4th Generation

(d) 6th Generation

Figure 16: Evolution of the population distribution for the minimisation of the 2D-RF using the ES algorithm with the default parameter configuration using seed 1. The algorithm converges to a local minimum at (467, 165).



(a) 1st Generation

(b) 2nd Generation

(c) 4th Generation

(d) 6th Generation

Figure 17: Evolution of the population distribution for the minimisation of the 2D-RF using the ES algorithm with the default parameter configuration using seed 2. The algorithm converges to a local minimum at (-281, -440).