4M17 Practical Optimisation

Coursework Assignment 2 – Optimisation Algorithm Performance Comparison

Candidate Number: 5628A

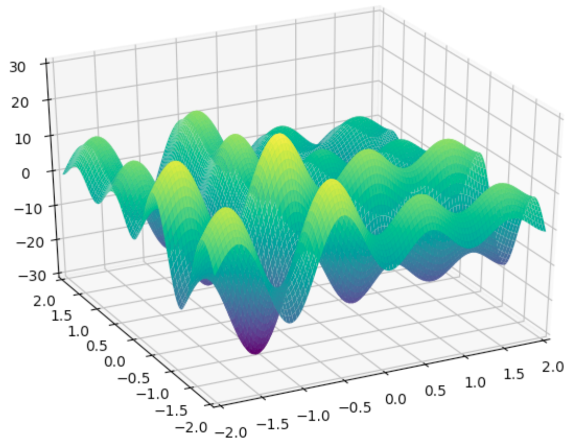
1 Introduction

In this coursework we evaluate and compare the performance of two optimisation algorithms, **Evolution Strategies** (ES) and **Tabu Search** (TS) on the 5D form of the Shubert's function (5D-SF):

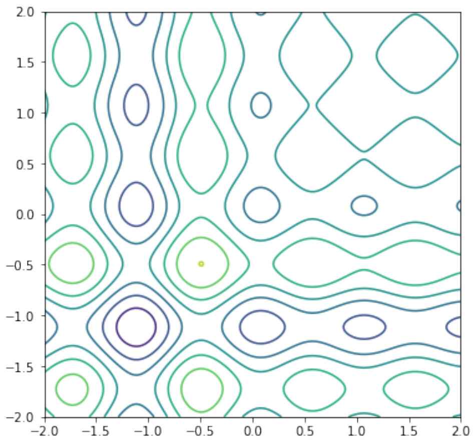
$$\min_{\mathbf{x}} \quad f(\mathbf{x}) = \sum_{i=1}^{n}\sum_{j=1}^{5} j\sin\left((j+1)x_i + j\right) \tag{1}$$
$$\text{subject to} \quad x_i \in [-2, 2] \quad \text{for} \quad i = 1, ..., n$$

ES and TS differ in many ways. For example, ES searches from one population of solutions to another whereas TS searches from one individual solution to another. Solution generation is probabilistic in ES but in TS it is deterministic (except in Search Diversification). One thing in common is that both ES and TS require only objective function information, not its derivative which greatly simplifies the computation.

The first part of this coursework investigates the effect on the performance of varying the control parameters or implementation options for each algorithm. After that, the two algorithms are compared in terms of performance, consistency and efficiency. Both algorithms are implemented from scratch using Python (see Appendix for code).



(a) 3D surface plot



(b) Contour plot

Figure 1: Visualisation of the 2D form of the Shubert's function

2 Evolution Strategies

ES is chosen over GA since it is designed for optimisation problems with continuous control variables. The representation of a solution includes the control variables $\mathbf{x}$, and *strategy parameters*: n variances σ_i^2 and $n(n-1)/2$ covariances σ_{ij}, which form the covariance matrix $\mathbf{\Sigma} = \{\sigma_{ij}\}$. The n-dimensional Gaussian distribution $\mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$ will be used in performing mutation. In our implementation, we work in terms of *rotation angles* α_{ij} instead of covariances σ_{ij}:

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

such that the result of rotations of $\mathbf{z}_c$, where $\mathbf{z}_c$ is the realisation of the *uncorrelated* n-dimensional Gaussian distribution with variances $\sigma_1^2, ..., \sigma_n^2$, by the $n(n-1)/2$ angles is equivalent to sampling from the *correlated* Gaussian distribution $\mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$ (Bäck 1995).

The algorithm is outlined below:

Algorithm 1: Evolution Strategy

```
initialise control variables $\mathbf{X} = \{\mathbf{x}_1, ..., \mathbf{x}_\lambda\}$ and strategy parameters $\mathbf{S} = \{(\boldsymbol{\sigma}_1, \boldsymbol{\alpha}_1), ..., (\boldsymbol{\sigma}_\lambda, \boldsymbol{\alpha}_\lambda)\}$
  where $\mathbf{x}_i \in \mathbb{R}^n$, $\boldsymbol{\sigma}_i \in \mathbb{R}^n$, $\boldsymbol{\alpha}_i \in \mathbb{R}^{n(n-1)/2}$;
while not converged do
    select $\mu$ parents;
    create $\lambda$ offspring by recombination followed by mutation;
    evaluate new population;
end
```

The control parameters include the population size λ, number of parents μ selected for mating, n initial standard deviations $\boldsymbol{\sigma}_0$, $n(n-1)/2$ initial rotation angles $\boldsymbol{\alpha}_0$, and constants τ, τ', β which control the sizes of individual mutations of the strategy parameters. There are also several implementation options such as different selection schemes and recombination methods.

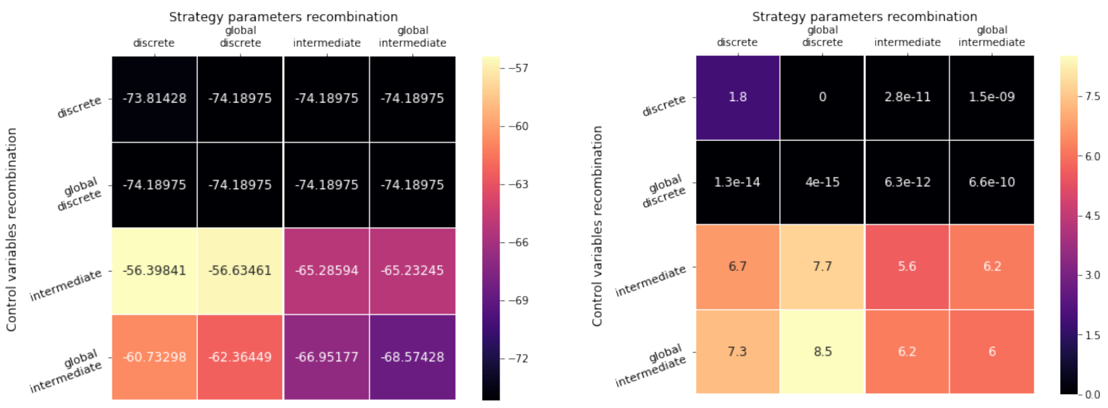
We begin with the following control parameters and implementation options and explore the effect of varying them:

- $\lambda = 140$, $\mu = 20$
- $\boldsymbol{\sigma}_0 = \sigma_0\mathbf{1}$ where $\sigma_0 = 0.1$
- $\boldsymbol{\alpha}_0 = \mathbf{0}$
- $\tau = \frac{1}{\sqrt{2\sqrt{n}}}$, $\tau' = \frac{1}{\sqrt{2n}}$, $\beta = 0.0873$
- (μ, λ)-selection

2.1 Recombination methods

In each iteration (generation), a new population of λ solutions is generated by recombination (and mutation) of the control variables $\mathbf{X}$ and strategy parameters $\mathbf{S}$ from the μ parents. Four recombination operators are outlined in Dr. Parks' lecture notes: discrete recombination, global discrete recombination, intermediate recombination and global intermediate recombination which can be applied to $\mathbf{X}$ and $\mathbf{S}$ separately (e.g. discrete for $\mathbf{X}$ and global intermediate for $\mathbf{S}$). Fixing the other control parameters, all $4 \times 4 = 16$ combinations for recombination are tested. For each combination, the ES is run 25 times using different initialisation of $\mathbf{X}$. We then take the mean and standard deviation of the best (lowest) objective function values found across the 25 runs.

Figure 2 shows that the performance of the ES depends mostly on the recombination method for $\mathbf{x}$; discrete or global discrete recombination for $\mathbf{x}$ generally gives a much better result than intermediate or global intermediate recombination. To have a better understanding, Figure 3 and Figure 4 visualise the optimisation processes for the 2D-SF problem using **(discrete, global discrete)** and **(intermediate, global discrete)** respectively, for ($\mathbf{x}$, $\mathbf{s}$) recombination. For (discrete, global discrete), the neighbourhood of the global



(a) Mean of best solutions

(b) Standard deviation of best solutions

Figure 2: Heatmaps of mean (a) and standard deviation (b) of the best solution found over 25 runs using different combinations of recombination methods for the control variables and strategy parameters.

minimum is quickly located in a few generations and the ES converges to the global minimum after tens of generations. However, for (intermediate, global discrete), the ES takes more generations to locate a neighbourhood of a local minimum and it converges to that local minimum instead of the global minimum. This occurs partly because of the objective function itself – Figure 5 shows why the algorithm could possibly end up in one of the local minima using intermediate recombination: at early stage, it is very likely that a solution near the global minimum and another solution near the second local minimum to the right of the global minimum, get paired up as parents. Taking the weighted average of those two parents, the resulting offspring (before mutation) would then end up very close to the local minimum right next to the global minimum since the spacing between minima is fairly uniform. In many other cases, even if one of the parents is near the global minimum, the offspring is "brought away" by the other parent and therefore it is generally hard to converge to a minimum using intermediate or global intermediate recombination operations on the control variables.

Except for (discrete, discrete), all ESs with discrete or global discrete recombination for $\mathbf{x}$ produce good results with high consistency (small standard deviation), especially for (discrete, global discrete), (global discrete, discrete) and (global discrete, global discrete) which all have a standard deviation of order 10^{-14} or less.

Figure 6 shows that the three recombination options have very similar rate of convergence. We can find the point of convergence by setting the convergence criteria to be the following:

$$|f_{worst} - f_{best}| \leqslant \epsilon_c = 10^{-12} \tag{3}$$

i.e. the absolute difference between the best and worst objective function values in the post-selection population. Table 1 shows that the option (global discrete, discrete) has the fastest convergence, but not by a large margin. (discrete, global discrete) requires a few hundred more evaluations but it achieves more consistent result.

Considering the best objective function value found, the consistency (standard deviation) and convergence rate, we conclude that **(discrete, global discrete)** recombination for ($\mathbf{x}$, $\mathbf{s}$) is the best option.

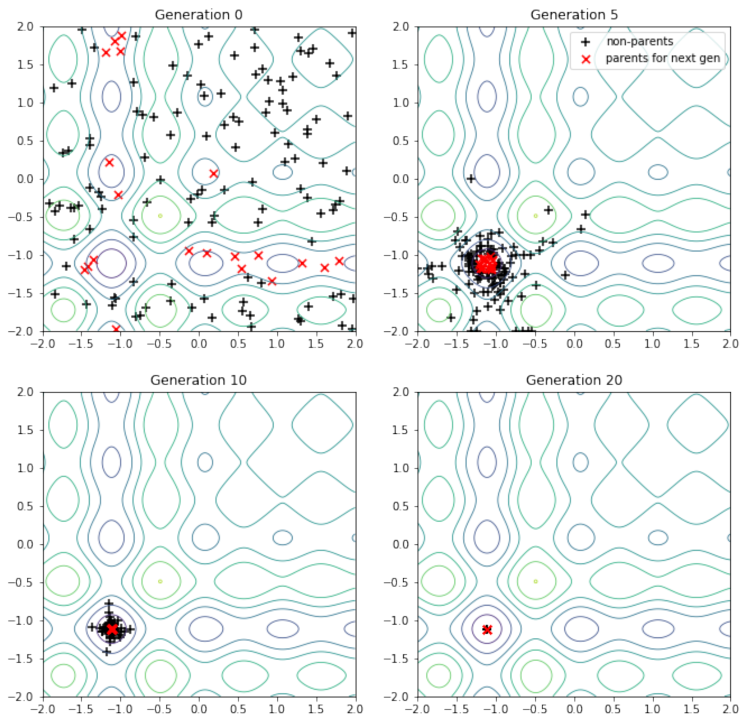


Figure 3: 2D visualisation of the ES on the 2D-SF using **discrete recombination** for control variables and **global discrete recombination** for strategy parameters

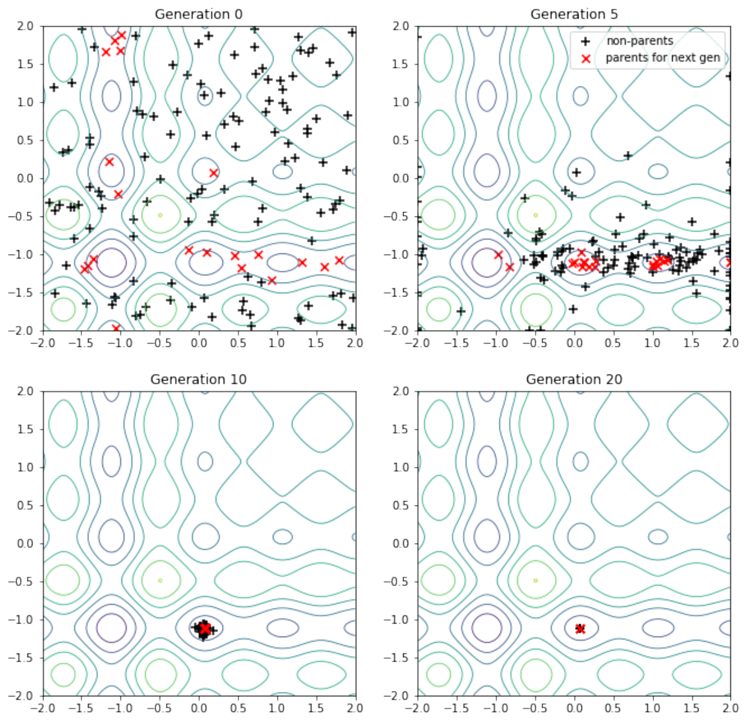


Figure 4: 2D visualisation of the ES on the 2D-SF using **intermediate recombination** for control variables and **global discrete recombination** for strategy parameters

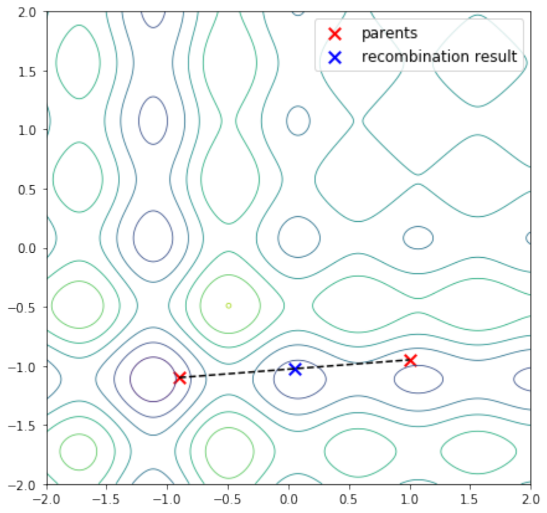


Figure 5: Example of an intermediate recombination which results in the new solution (before mutation) being close to a local minimum

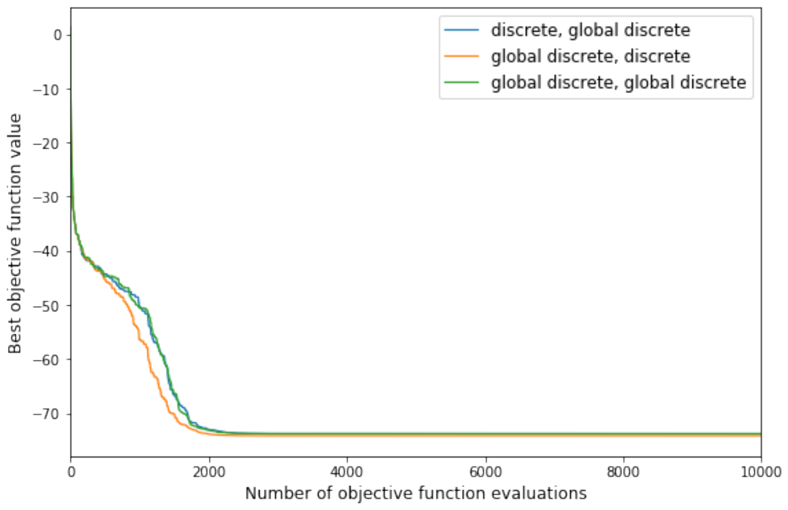


Figure 6: Best objective function value averaged over 25 runs for ESs with different combination of recombination operations for $\mathbf{x}$ and $\mathbf{s}$

($\mathbf{x}$, $\mathbf{s}$) recombination operators	Mean # evaluations for convergence
(discrete, global discrete)	8472.8
(global discrete, discrete)	7789.6
(global discrete, global discrete)	8103.2

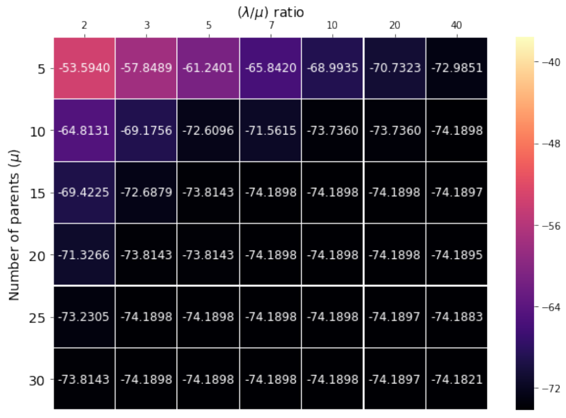
Table 1: Average number of evaluations required for convergence for different combination of $\mathbf{x}$ and $\mathbf{s}$ recombination operations

2.2 Population size and number of parents

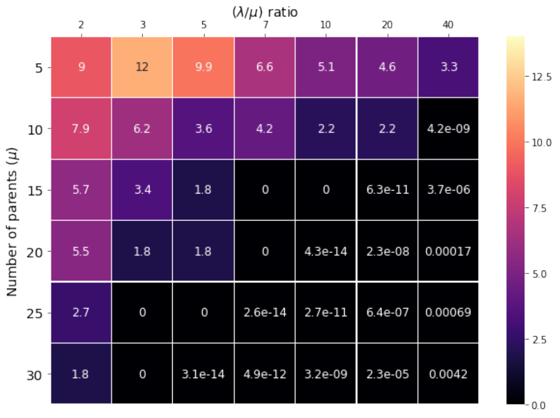
In each generation, μ "fittest" parents are selected from λ solutions in the population to generate λ new solutions. The choice of population size λ and number of parents μ could affect the performance of the ES. For small λ's, each iteration would require fewer objective function evaluations (in fact # objective function evaluations per generation $= \lambda$) and hence the algorithm could go through more generations while having the same number of objective function evaluations, which could potentially increase the convergence rate w.r.t. the number of evaluations. Conversely, if λ is too big, each generation would require many objection function evaluations, reducing the convergence rate. However, if λ is not big enough, the ES might not be able to fully explore the search space and end up at some local minimum, especially in high-dimensional space.

Similar, if μ is too small, the ES might converge too quickly and get stuck in a local minimum before even exploring the search space thoroughly. This makes the algorithm less consistent since it would depend more on the initialisation of $\mathbf{x}$. On the other hand, if μ is too large, the ES would take longer to converge since "weaker" parents are selected for mating at each generation. We therefore want to find the best combination of (λ, μ) that produces consistently good results with a reasonable convergence rate.

Instead of working directly with μ and λ, we search for the best combination in term of μ and λ/μ ratio. Again, fixing the control parameters, we run the algorithm 25 times and take the mean and standard deviation of the best objection function values found, for each combination of μ and λ/μ.



(a) Mean of best solutions



(b) Standard deviation of best solutions

Figure 7: Heatmaps of mean (a) and standard deviation (b) of the best solution found over 25 runs using different combinations of μ and λ/μ ratio

The results shown in Figure 7 agree with our prediction above. If either μ or λ/μ is too small, the ES would easily get stuck at some local minimum and would not be able to locate the global minimum. As μ and λ/μ increase to some certain values, the ES becomes stable, producing very consistent results (e.g. $\mu, \lambda/\mu =$ (25, 5), (20, 7), (15, 10)) with standard deviation of virtually zero for a precision of $\sim 10^{-16}$. However, if we further increase μ or λ, the standard deviation starts to increase because as explained above, although the ES is almost always able to locate the global minimum, the convergence rate is reduced due to the large population or number of parents. Figure 8 and Figure 9 further illustrate these points.

For those in Figure 7 that have the best objective function value (-74.1898) and standard deviation (0), the convergence times (in terms of number of evaluations) are recorded in Table 2 and it shows that $\mu = 15$ and $\lambda/\mu = 7$ achieves the best result with the fastest convergence rate.

μ	λ/μ	λ	Mean # evaluations for convergence
15	10	150	7199.7
15	7	105	6090.3
20	7	140	7981.6
25	5	125	7748.0
25	3	75	6375.6
30	3	90	7293.1

Table 2: Number of objective function evaluations required for convergence for different combination of μ and λ. The convergence criteria is stated in (3).

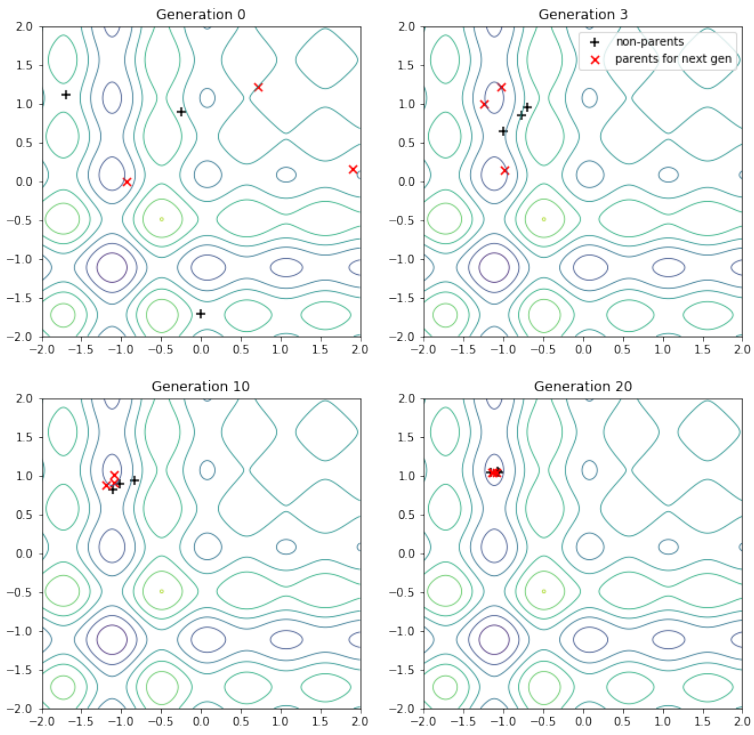


Figure 8: Minimisation of 2D-SF by an ES with population size $\lambda = 6$ and number of parents $\mu = 3$. This extreme case is chosen because in 2D the search space is much smaller than in 5D. The figure shows that the ES is not able to explore the space thoroughly to find the global minimum before converging to a local minimum because of the small population size. In 5D, an even larger population is needed since the search space is much bigger.

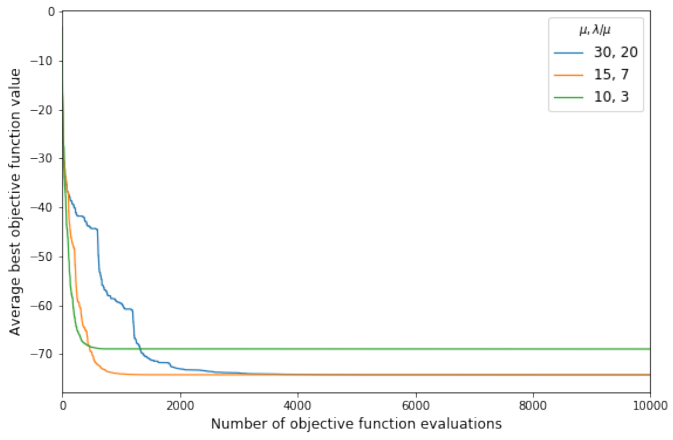


Figure 9: Plot of average best objective function value vs number of evaluations for three ESs of different sets of $(\mu, (\lambda/\mu))$. Both the orange line (15, 7) and blue line (30, 20) are always able to find the global minimum but the blue line converges slower than orange due to its large population. The green line (10, 3) converges faster than the orange line but it often finds a suboptimal solution.

2.3 Mutation

In an ES, mutation is done by first mutating the strategy parameters $\boldsymbol{\sigma}$ and $\boldsymbol{\alpha}$ which form the covariance matrix $\boldsymbol{\Sigma}$. Then, $\mathbf{x}$ is mutated using the new strategy parameters:

$$\mathbf{x}' = \mathbf{x} + \mathbf{n} \tag{4}$$

where $\mathbf{n} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$ which can be generated by rotating an uncorrelated Gaussian random vector $\mathbf{n}_u$ by the $n(n-1)/2$ angles:

$$\mathbf{n} = \left(\prod_{i=1}^{n-1} \prod_{j=i+1}^{n} \mathbf{R}(\alpha_{ij}) \right) \mathbf{n}_u \tag{5}$$

$$\mathbf{n}_c \sim \mathcal{N}\left(\mathbf{0}, \mathrm{diag}\left(\sigma_1^2, ..., \sigma_n^2\right)\right)$$

where $\mathbf{R}(\theta)$ is the rotation matrix corresponding to rotation angle θ.

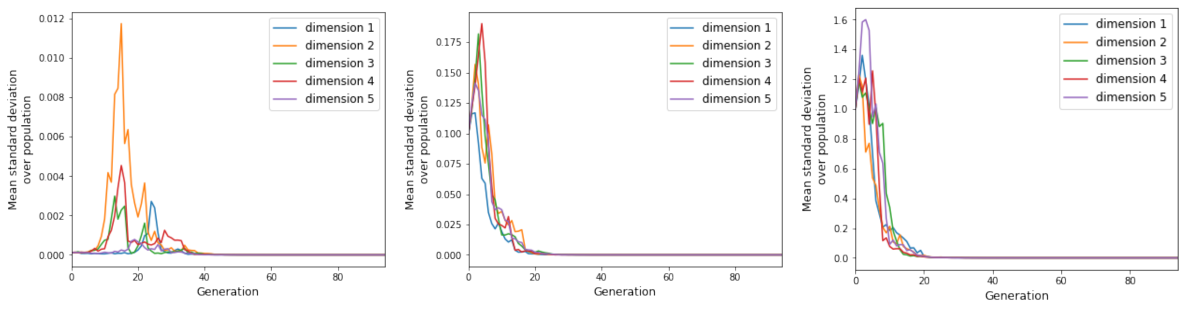
In this type of ES, the strategy parameters are also subject to recombination and mutation. This process is known to be *self-adaptive* in the sense that the algorithm evolves its own strategy parameters appropriately as the search progresses, instead of having a preset schedule.

2.3.1 Initial strategy parameters

Since the population is initialised by randomly sampling from a uniform distribution within the search space, it makes sense to set the initial Gaussian distribution for $\mathbf{x}$ mutation to be uncorrelated, i.e., $\boldsymbol{\alpha}_0 = \mathbf{0}$ and $\boldsymbol{\Sigma}$ is a diagonal matrix. Moreover, all the control variables have identical scales and hence all the standard deviations are initialised to the same value: $\boldsymbol{\sigma}_0 = \sigma_0 \mathbf{1}$, $\boldsymbol{\Sigma} = \sigma_0^2 \mathbf{I}$. Therefore, the only initial parameter to adjust is σ_0.

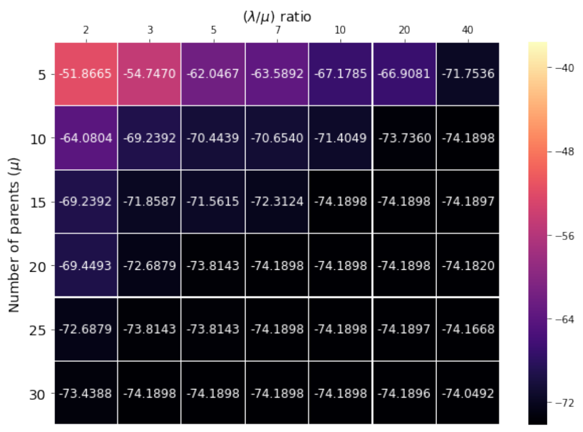
Figure 11 shows that for fixed (μ, λ), the performance of the ES is less adequate when σ_0 is too small (0.0001) or too large (1.0). For each member in the population, its strategy parameters essentially represent the *mutation strengths* in different directions of the search space. Figure 10 shows that the standard deviations generally increase at the early stage of the ES, when it is exploring the search space, and later reduce down to zero when the neighbourhood of a minimum is located. For $\sigma_0 = 0.0001$, the small initial standard deviations do not allow the ES to explore the search space thoroughly; it takes around 20 generations to increase its mutation strength to over 100 times the initial standard deviation. During these 20 generations, it is very likely that the population is already concentrated around some local minimum since many rounds of selection have taken place. This is illustrated in Figure 12. On the other hand, if σ_0 is large, even if an offspring has located the neighbourhood of the global minimum, it is forced to move away (in the scale of ~ 1) by mutation, causing the ES to be less stable.

σ_0 in the order of 0.01-0.1 gives the best result. We will proceed with $\sigma_0 = 0.1$.

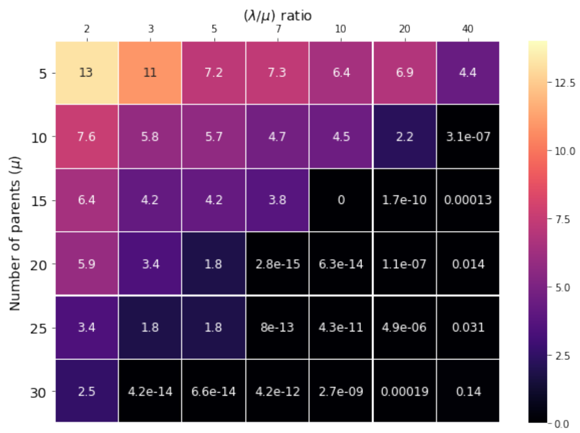


(a) $\sigma_0 = 0.0001$ (b) $\sigma_0 = 0.1$ (c) $\sigma_0 = 1$

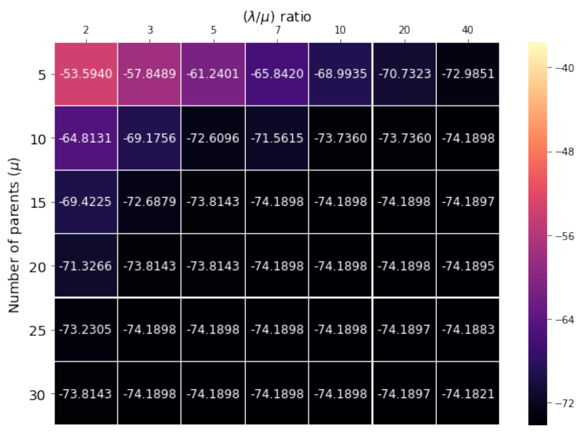
Figure 10: Mean of standard deviations (strategy parameter) over the population for each control variable, using different initial standard deviation ($\sigma_0 = 0.0001, 0.1, 1$). For any σ_0, the standard deviations generally increase in the early stage of the ES when it is exploring the search space to locate a minimum (local or global).



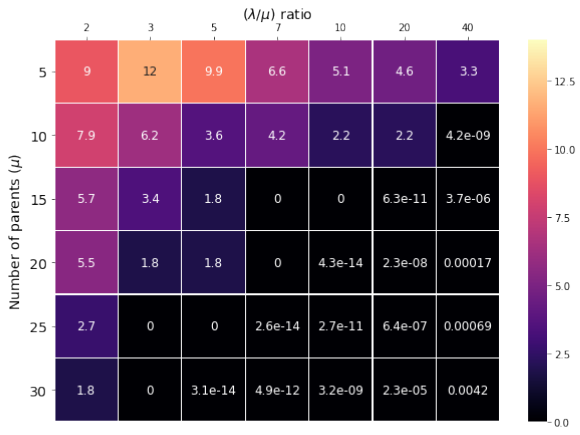
(a) Mean for $\sigma_0 = 0.0001$



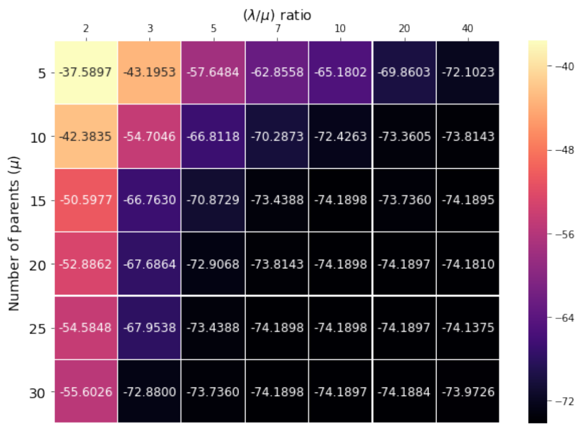
(b) Standard deviation for $\sigma_0 = 0.0001$



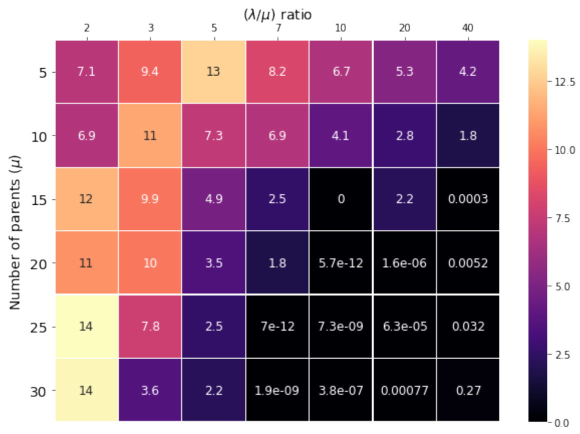
(c) Mean for $\sigma_0 = 0.1$



(d) Standard deviation for $\sigma_0 = 0.1$



(e) Mean for $\sigma_0 = 1$



(f) Standard deviation for $\sigma_0 = 1$

Figure 11: Mean and standard deviation of the best objective function value found over 25 runs using different initial strategy parameters ($\sigma_0 = 0.0001, 0.1, 1$)

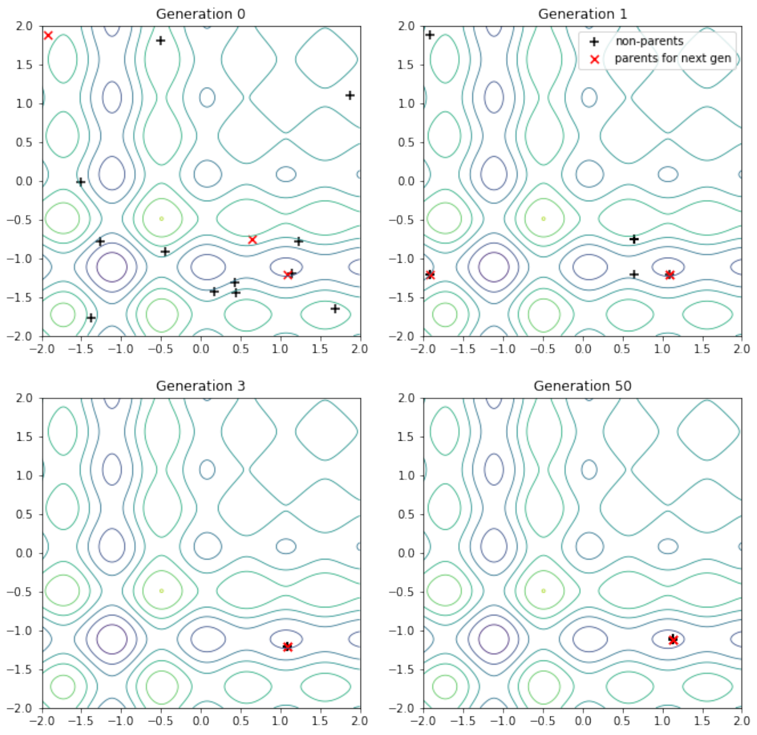


Figure 12: Minimisation of 2D-SF using an ES with a very small initial standard deviation. The initial mutation strength is too small for the ES to explore the space thoroughly, resulting in the ES converging to a local minimum before being able to increase the mutation strength sufficiently. Again, a small population is chosen to illustrate this behaviour since the 2D search space is much smaller than the 5D search space.

2.3.2 Mutation of rotation angles

It is possible for the ES to not mutate the rotation angles so that the covariance matrix $\mathbf{\Sigma}$ remains diagonal and we essentially draw random vectors from an **uncorrelated** n-dimensional Gaussian during the mutation of $\mathbf{x}$. In other words, self-adaption on $\boldsymbol{\sigma}$ alone might be sufficient for the ES to arrive at the global minimum.

Table 3 compares the performance of the ESs on the 5D-SF with and without angle mutation. Both ESs are able to consistently find the global minimum within 10000 evaluations. Although the ES without angle mutation on average requires 5% more objective function evaluations, the CPU time needed for convergence is significantly reduced by 28%. This is because during mutation of $\mathbf{x}$, the rotation of the random vector $\mathbf{n}_c$ by the $n(n-1)/2$ angles (see (5)) requires $\mathcal{O}(n^2)$ matrix multiplication operations, and this is done λ times for every generation, making it one of the most computationally heavy tasks in the algorithm. Figure 13 shows that the reduction in CPU time increases with the number of control variables, by using just an uncorrelated Gaussian for mutation of $\mathbf{x}$, i.e. zero rotation angles throughout. Therefore, not mutating the rotation angles and leaving them as zeros throughout the algorithm would require slightly more objective function evaluations but overall the CPU time reduces. Ultimately, the computational efficiency of an algorithm is judged on the running time; thus, not mutating the rotation angles is the better option.

Mutate angles	**Best objective function value**		**# objective function evaluations for convergence**		**CPU time (s) for convergence**	
	Mean	Std dev.	Mean	Std dev.	Mean	Std dev.
Yes	-74.18975	0	5770.8	221.21	2.223	0.0866
No	-74.18975	0	5875.8	388.90	1.353	0.0921

Table 3: Comparison of performance and convergences time for ESs on the 5D-SF with and without angle mutation

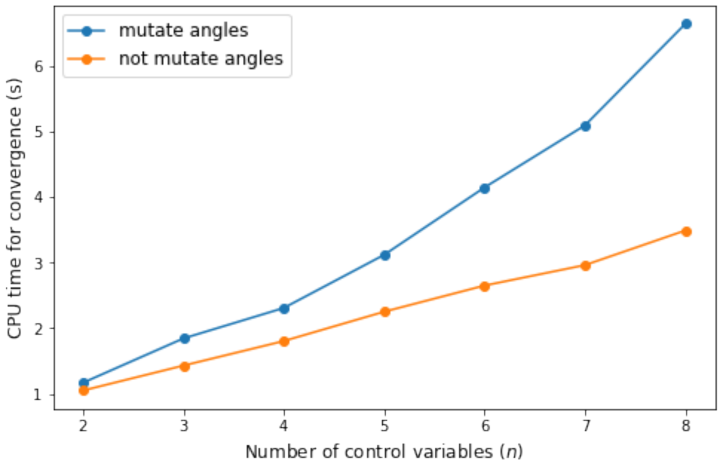


Figure 13: Plot of CPU time required for convergence vs the number of control variables

2.4 Selection Scheme

Two selection schemes are outlined in Dr Parks' notes: (μ, λ)-selection and $(\mu + \lambda)$-selection. Table 4 shows that the performance of ESs using the two different selection schemes differ by very little, with (μ, λ)-selection having a slightly better performance in terms of number of objective function evaluations and CPU time. The longer CPU time for $(\mu + \lambda)$-selection is mostly due to the fact that more elements (objective function values) need to be sorted during the selection step.

Selection scheme	**Best objective function value**		**# objective function evaluations for convergence**		**CPU time (s) for convergence**	
	Mean	Std dev.	Mean	Std dev.	Mean	Std dev.
(μ, λ)	-74.18975	0	5875.8	388.90	1.386	0.0940
$(\mu + \lambda)$	-74.18975	0	6283.2	282.81	1.492	0.0776

Table 4: Comparison of performance and convergences time for ESs on the 5D-SF using different selection schemes

2.5 ES conclusion

The following parameters and implementation options for ES are chosen:

- Global discrete recombination for control variables

- Discrete recombination for strategy parameters
- $\mu = 15$ and λ/μ ratio $= 7$, i.e., $\lambda = 105$
- Initial strategy parameters: $\sigma_0 = 0.1, \alpha_0 = 0$
- Rotation angles are not mutated, i.e., the Gaussian distribution for mutation of $\mathbf{x}$ remains uncorrelated
- (μ, λ)-selection scheme

The objective function values minimised using an ES with this set of parameters has a mean of -74.18975 and standard deviation of 0 (for a precision of $\sim 10^{-16}$). The best individual solution has objective function value of -74.18975 at $\mathbf{x} = [-1.11410, 1.11410, 1.11410, 1.11410, 1.11410]^T$.

3 Tabu Search

TS attempts to solve a hard optimisation problem by imposing restrictions to help the search escape from local minima. TS has three main components: **local search**, **Search Intensification** and **Search Diversification** which are controlled by the *Short Term Memory* (STM), *Medium Term Memory* (MTM) and *Long Term Memory* (LTM) respectively. Additionally, **Step Size Reduction** is carried out to carefully search the neighbourhood of the best solution found at the time.

Early investigation shows that TS is rather sensitive to the control parameters and is generally less consistent than ES given the same number of objective function evaluations. Therefore, the number of runs for each test is increased to 50 for more reliable results.

The following initial control parameters are used:

- STM size $N = 3$
- MTM size $M = 4$
- Number of sectors per control variable $K = 7$
- Initial step size $= 1$
- SSR factor $r = 0.5$

3.1 Local search

The movement during a local search is restricted by the STM (tabu) which stores the last N locations successfully visited. A move is rejected if it is in the STM or if it violates some constraint(s). During a local search, even if the algorithm has located a local minimum, it is forced by the STM to climb out and explore further.

The main control parameter for local search is the size of the STM, N. A range of values for N are tested while keeping the other control parameters constant. Figure 14 shows that the performance improves as N increases up to 26, beyond which further increase in N results in similar or worse performance. A relatively large N is preferred for this 5D-SF problem compared to the number of control variables. A possible reason is that it is a difficult optimisation problem to solve with many local minima. During a local search, the TS could get stuck between two local minima if N is not big enough. A large STM could help the TS escape by imposing stronger restriction on the search movement and prevent the TS from going back and forth between two local minima. This is illustrated in Figure 15.

Beyond $N = 26$, further increase in N does not improve the performance and Figure 16 shows that the behaviours and convergence velocities of the TSs with large N are very similar. Therefore, $N = 26$ is chosen.

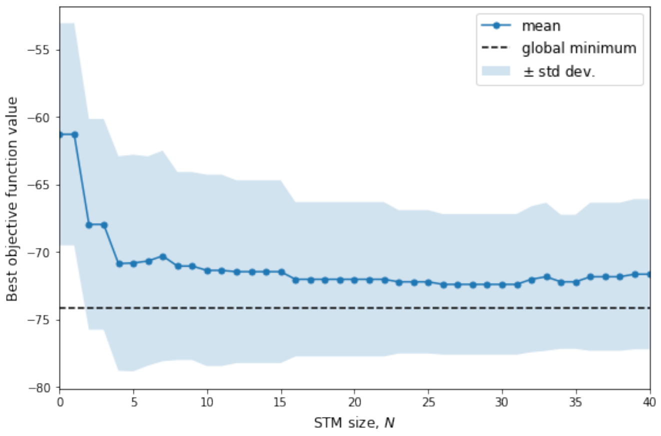
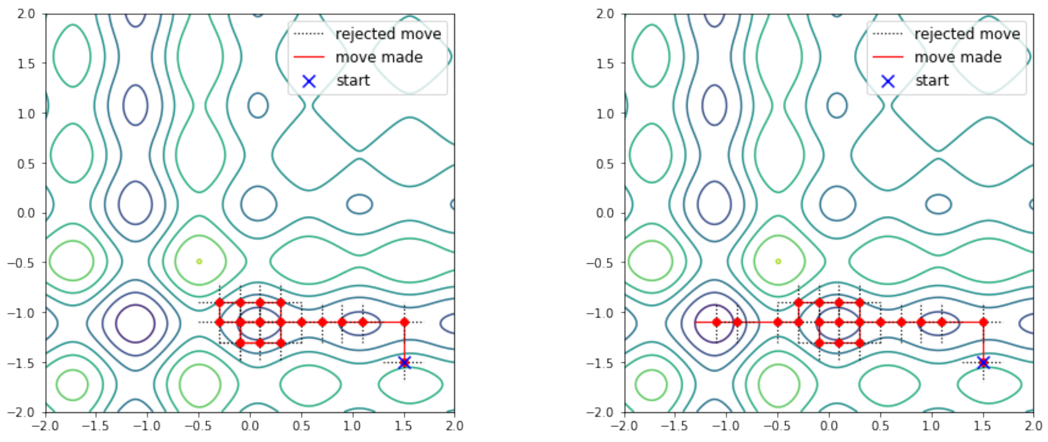


Figure 14: Mean and standard deviation of the best objective function value found over 50 runs using different STM sizes



(a) $N = 5$ (b) $N = 10$

Figure 15: Examples of local search run for 25 iterations (moves) on the 2D-SF starting at (1.5, -1.5), using $N = 5$ and $N = 15$. For $N = 5$, the local search is stuck between two local minima, going back and forth between them. For $N = 15$, it is able escape from the local minima by disallowing itself from moving back to the two minima, without the help of SI or SD.

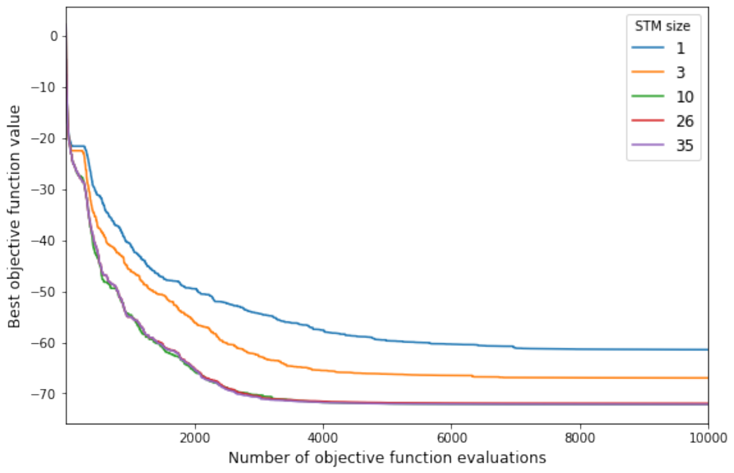


Figure 16: Plot of best objective function value (averaged over 50 runs) vs number of evaluations for different STM sizes

3.2 Search Intensification (SI)

During the search, the M best solutions are stored in MTM and updated every time a location is visited. If the local search produces no improvement in the best solution for a specific number of iterations (10 in our case), the search is *intensified*: the search location is moved to the average position in the MTM, i.e., an "average best" position.

Again, a range of values for M are tested (Figure 17) and the result shows that $M = 4$ gives the best performance.

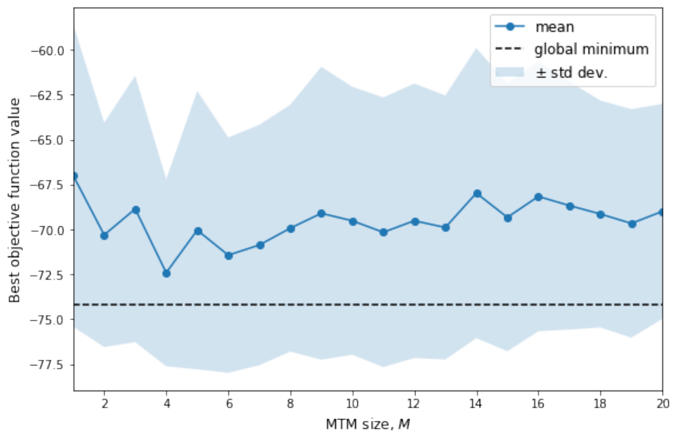


Figure 17: Mean and standard deviation of the best objective function value found over 50 runs using different MTM sizes

3.3 Search Diversification (SD)

The search space is divided into K^n uniform sections (i.e. K sectors per dimension). The LTM keeps track of whether a sector has already been "reasonably searched". We define the *search thoroughness* of a sector to be the number of locations successfully visited in that sector, and the threshold for “reasonably searched” to be a thoroughness of 10.

If after SI, local search does not produce a new best solution for another 5 iterations, the search is *diversified*. The search location is moved to a randomly selected sector which has not been thoroughly searched.

The effect of the number of sectors per control variable, K is investigated by varying the value and observing the performance. Figure 18 shows that changing K would not result in a dramatic improvement/reduction in performance for this optimisation problem. One possible reason could be that the 5D search space is very large; if we have K sectors per control variable, the total number of sectors would be K^n. For $K = 4$, it would have 1024 sectors and for $K = 10$, it would have 10^5 sectors. Looking at a plot of objective function vs number of evaluations for a typical TS (Figure 20), SD happens only $\sim$ 15 times in 10000 evaluations and it is almost always unable to improve the best solution, since it is very hard for SD to locate a “good” sector by randomly choosing from that many sectors.

Figure 19 shows that K has very little effect on the convergence rate. The best value ($K = 5$) is chosen.

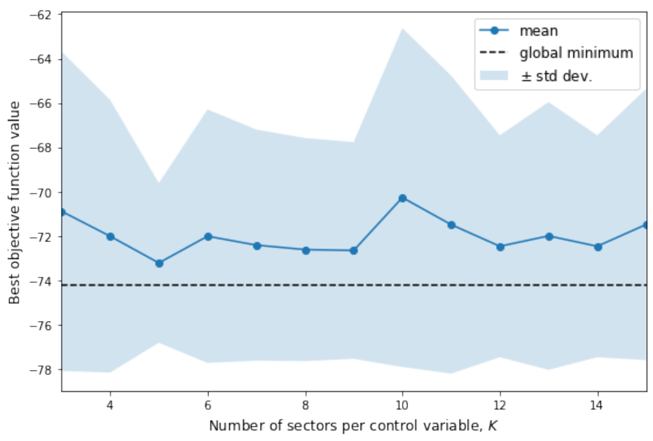


Figure 18: Mean and standard deviation of the best objective function value found over 50 runs using different number of sectors per control variable

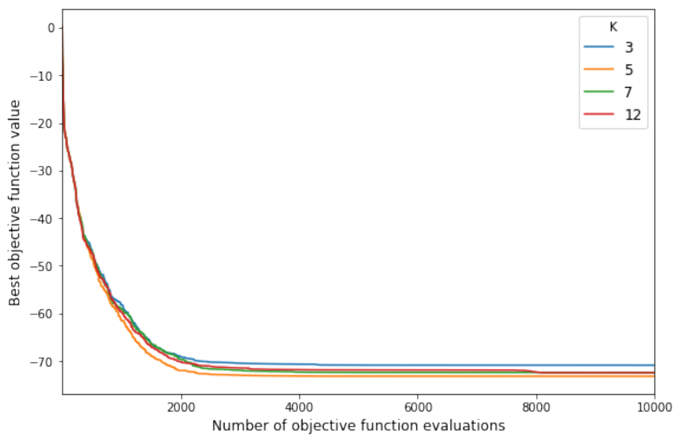


Figure 19: Best objective function value (averaged over 50 runs) vs number of evaluations for different values of K (number of sectors per control variables)

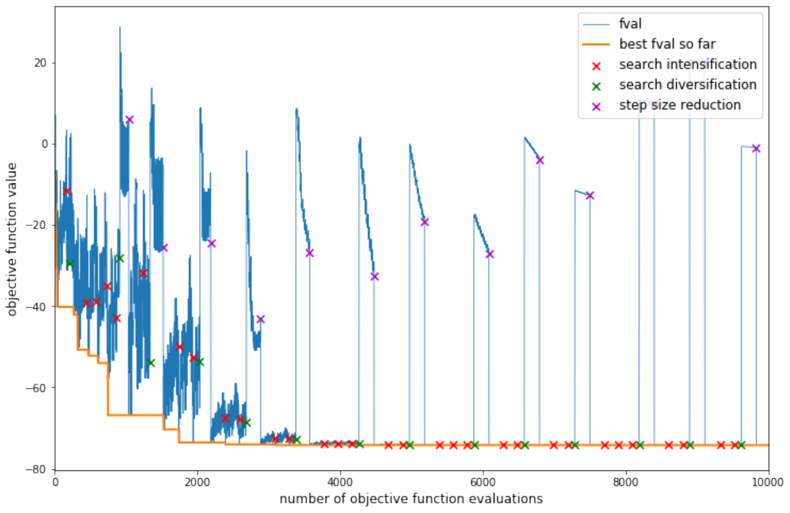
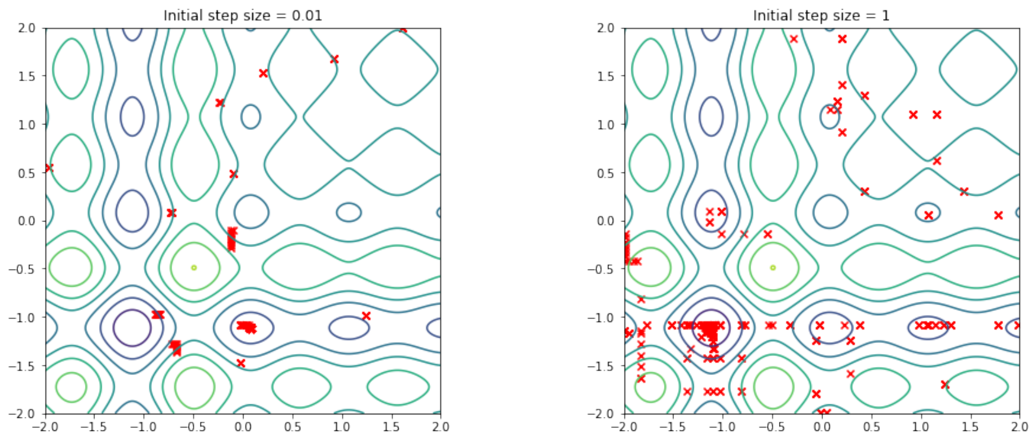


Figure 20: Example plot of objective function value vs number of evaluations for a TS

3.4 Initial step size

Having a large (e.g. 1) initial step size would help the algorithm explore different parts of the search space at the early stage. Figure 21 illustrate the difference between having a large and small initial step size: for small step size, even if SD or SI has brought the TS to the neighbourhood of the global minimum, the step size is so small that it cannot descent close enough to the minimum within limited local search iterations, and ends up being brought away by SD/SSR. Figure 22 shows that an initial step size of 1 has the best convergence rate and objective function value.



(a) Initial step size = 0.01

(b) Initial step size = 1

Figure 21: Locations visited of the TS using different initial step sizes on the 2D-SF for 10000 function evaluations. For initial step size = 0.01, even though SD has brought it to the neighbourhood of the global minimum, the step size is already too small and the TS returns to the local minimum after 10 local search iterations.

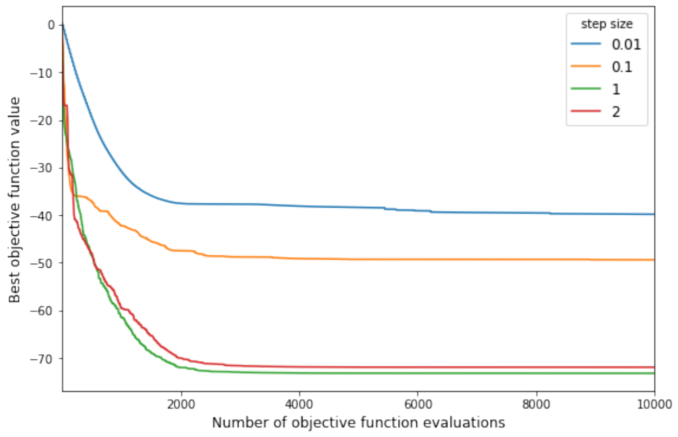


Figure 22: Objective function value plot for different initial step sizes

3.5 Step Size Reduction (SSR) factor

If both SI and SD are not able to produce an improved best solution, the step size is reduced and the search location is brought to the current best solution so that the neighbourhood of the best solution can be more carefully searched. In SSR, the step size is multiplied by the *SSR factor*, r: $\delta' = r\delta$. Figure 23 shows that the effect of r on the performance is rather unpredictable and does not have a general trend. In theory, if r is too small, the step size might be reduced too quickly resulting in the same problem as illustrated in Figure 21(a). The major disadvantage of having a large r is the reduced convergence velocity. $r = 0.8$ produces the best performance with a mean best objective function value of -74.126 and standard deviation of 0.0456.

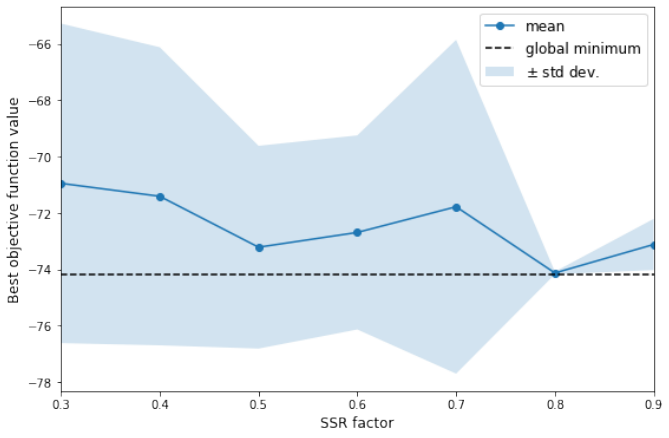


Figure 23: Mean and standard deviation of the best objective function value found over 50 runs using different SSR factor

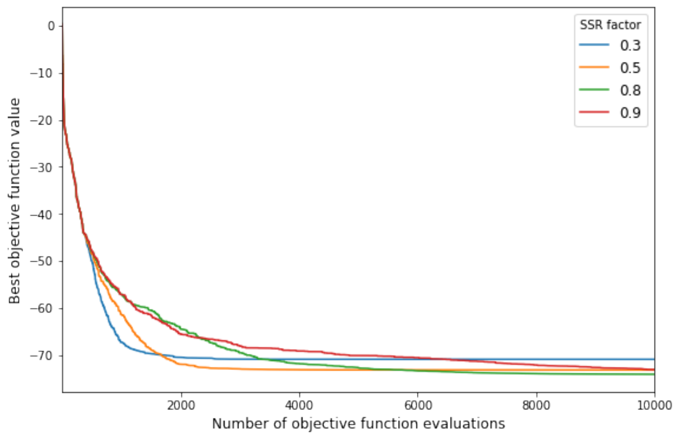


Figure 24: Best objective function value (averaged over 50 runs) vs number of evaluations for different SSR factor r. Generally for increasing r, the convergence rate decreases but the best solution found improves.

3.6 TS Conclusion

The following parameters for TS are chosen:

- STM size, $N = 26$
- MTM size, $M = 4$
- Number of sectors per control variable, $K = 5$
- Initial step size $= 1$
- SSR factor $= 0.6$

With this set of parameters, the objective function values minimised using the TS has a mean of -74.12585 and standard deviation of 0.0456. The best individual solution has objective function value of -74.18386 at $\mathbf{x} = [-1.10919 - 1.11138 - 1.11385 - 1.11365 - 1.11268]^T$.

4 Comparison of Algorithms

Algorithm	**Best individual objective function value**	**Best objective function value**		**CPU time (s) for 10000 evaluations**	
		Mean	Std dev.	Mean	Std dev.
ES	-74.18975	-74.18975	0	1.793	0.0993
TS	-74.18386	-74.12585	0.0456	0.285	0.0102

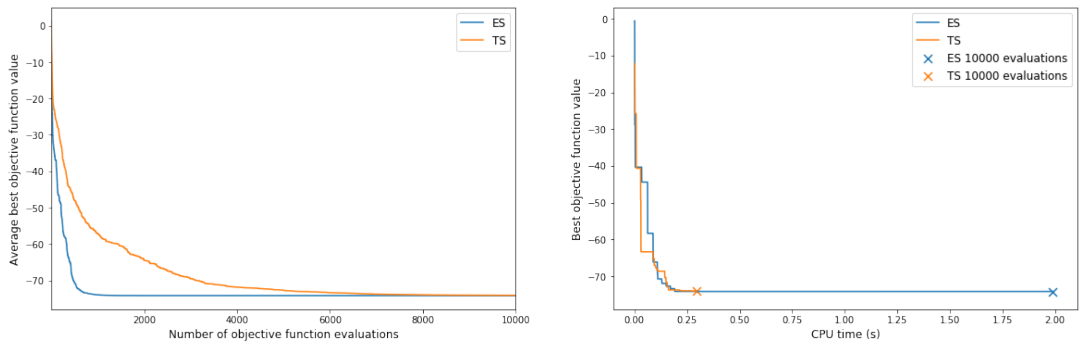
Table 5: Comparison of performance of ES and TS

Fixing the number of objective function evaluations at 10000, the performance of Evolution Strategy is far superior to Tabu Search, with better (lower) minimum objective function value and very small standard deviation. However, fixing the number of evaluations is not a fair way to compare the algorithms. Table 5 shows that ES has a much higher *computational cost per evaluation* as it has a few other steps that requires significant computation time such as mutation and selection. Figure 25 shows that although ES's convergence rate w.r.t. the number of evaluations is much faster than TS's, their convergence rates w.r.t. CPU running time are similar. If we instead fix the CPU time and run the algorithms, Table 6 shows that TS is able to achieve a standard deviation of $\sim 10^{-7}$ after 1.0 s, and its performance is comparable with ES after a running time of $\sim$ 2.0 s, although the number of evaluations is way more than the allowed limit. ES still achieves better results on average for up to 2 s of running time.

The running times for both algorithms can be reduced by parallelising the computation steps. In ES, the λ evaluations of objective function for the entire population and the production of offspring by mutation and recombination, can both be parallelised; in TS, the $2n$ objective function evaluations in a local search iteration can be also be done simultaneously. The reduction in running time by parallelisation would depend on the number of processors available.

ES is an population-based approach which devotes more time sampling points around the search space in the beginning and gradually converges to a minimum. TS devotes more effort on regions that have good solutions and it proceeds to a local minimum much more aggressively than ES (Figure 26).

One drawback of TS is its sensitivity to the control parameters. For instance, SSR factors of 0.7 and 0.8 would lead to very different results. Its control parameters also tend to be more problem-specific (e.g. a very large STM size N is suitable for this problem). TS therefore requires more fine tuning on its control parameters whereas control parameters in ES are generally more robust to varying problems.

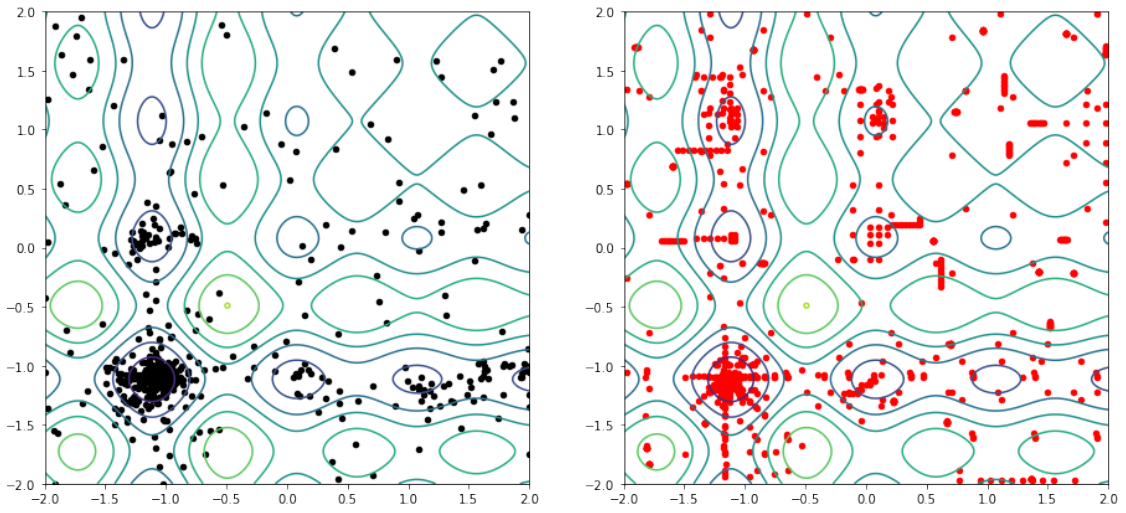


(a) (b)

Figure 25: Plots of best objective function value against (a) number of evaluations and (b) CPU time. Panel (b) shows only a single run for each algorithm.

CPU time (s)	ES # evaluations	ES Best function value		TS # evaluations	TS Best function value	
	Mean	Mean	Std dev.	Mean	Mean	Std dev.
0.3	1589	-74.17998	9.30×10^{-3}	8691	-74.05239	1.26×10^{-1}
0.5	2654	-74.18973	3.96×10^{-5}	15311	-74.18721	2.22×10^{-3}
1.0	5316	-74.18975	4.27×10^{-9}	31863	-74.18975	2.72×10^{-7}
1.5	7967	-74.18975	1.73×10^{-14}	48450	-74.18975	1.43×10^{-11}
2.0	10601	-74.18975	1.42×10^{-14}	65174	-74.18975	1.45×10^{-14}

Table 6: Comparison of performance of ES and TS under **fixed CPU time**



(a) Evolution Strategy

(b) Tabu Search

Figure 26: Searched locations for ES (a) and TS (b) on 2D-SF (10000 evaluations)

References

Bäck, T. (1995). *Evolutionary Algorithms in Theory and Practice*. Oxford University Press.

Appendix

A.1 Evolution Strategy – Python Code

evolution_strategy.py

```python
import numpy as np
from scipy import stats
from time import time

n_fvals = 0
best_fval_history = []
timestamps = []
start_time = 0


def recombination(X, method='discrete', n_offspring=140):
    if method == 'discrete':
        return discrete_recombination(X, n_offspring=n_offspring)

    elif method == 'global discrete':
        return global_discrete_recombination(X, n_offspring=n_offspring)

    elif method == 'intermediate':
        return intermediate_recombination(X, n_offspring=n_offspring)

    elif method == 'global intermediate':
        return global_intermediate_recombination(X, n_offspring=n_offspring)

    else:
        raise ValueError('unknown recombination method')


def discrete_recombination(X, n_offspring=140):
    population, n = X.shape
    offspring = np.zeros((n_offspring, n))
    for i in range(n_offspring):
        # randomly select 2 parents
        parent1, parent2 = np.random.randint(population, size=2)
        while parent1 == parent2:
            parent2 = np.random.randint(population)
        # randomly determine which parent contribute to which component
        coin_tosses = np.random.choice([True, False], n)

        # recombine
        offspring[i, coin_tosses] = X[parent1, coin_tosses]
        offspring[i, ~coin_tosses] = X[parent2, ~coin_tosses]

    return offspring


def global_discrete_recombination(X, n_offspring=140):
    population, n = X.shape
    offspring = np.zeros((n_offspring, n))

    for i in range(n_offspring):
        # randomly select parent for each component by balanced roulette wheel
        parents = np.random.randint(population, size=n)
        # recombine
        offspring[i, :] = X[parents, range(n)]

    return offspring


def intermediate_recombination(X, n_offspring=140, w=0.5):
    population, n = X.shape
    offspring = np.zeros((n_offspring, n))

    for i in range(n_offspring):
        # randomly select 2 parents
        parent1, parent2 = np.random.randint(population, size=2)
        while parent1 == parent2:
            parent2 = np.random.randint(population)

        # recombine by weighted average
        offspring[i, :] = w * X[parent1] + (1 - w) * X[parent2]

    return offspring


def global_intermediate_recombination(X, n_offspring=140, w=0.5):
    population, n = X.shape
    offspring = np.zeros((n_offspring, n))

    for i in range(n_offspring):
        for j in range(n):
            # randomly select 2 parents
            parent1, parent2 = np.random.randint(population, size=2)
            while parent1 == parent2:
                parent2 = np.random.randint(population)
            # take weighted average for that component
            offspring[i, j] = w * X[parent1, j] + (1 - w) * X[parent2, j]

    return offspring


def mutation(X, SP, tau, tau_prime, beta, alpha_indices, mutate_angles=True):
    n = X.shape[1]
    S = SP[:, :n].copy()    # standard deviations
    A = SP[:, n:].copy()    # rotation angles
    X_new = np.zeros_like(X)

    for i in range(X.shape[0]):
        # mutate sigma and alpha
        rv = np.random.randn(SP.shape[1] + 1)
        S[i] = S[i] * np.exp(tau_prime * rv[0] + tau * rv[1:n+1])
        if mutate_angles:
            A[i] = A[i] + beta * rv[n+1:]
        SP_new = np.concatenate((S, A), axis=-1)

        # mutate x
        # sample uncorrelated Gaussian
        dx = stats.multivariate_normal.rvs(np.zeros(n), np.diag(np.square(S[i])))
        if mutate_angles:
            # rotate dx by the n(n-1)/2 angles
            for alpha_i, alpha in enumerate(A[i]):
                # construct rotation matrix
                R = np.eye(n)
                ri, rj = alpha_indices[alpha_i]
                R[(ri, rj, ri, rj), (ri, rj, rj, ri)] = \
                    [np.cos(alpha), np.cos(alpha), -np.sin(alpha), np.sin(alpha)]
                # rotate dx
                dx = np.dot(R, dx)

        X_new[i] = X[i] + dx

    return X_new, SP_new


def evolution_strategy(obj_func, n, population=140, n_parents=20, s0=0.1,
                       x_recombination_method='discrete',
                       sp_recombination_method='intermediate',
                       select_parents=False, mutate_angles=True, epsilon=1e-12,
                       max_n_fvals=1e4):
```

```python
    try:
        reset()

        X = np.random.uniform(-2, 2, size=(population, n))  # randomly sample in the space
        S0 = s0 * np.ones((population, n))      # initial std
        A0 = np.zeros((population, int(n*(n-1)/2)))       # initial rotation angles
        SP = np.concatenate((S0, A0), axis=-1)      # strategy parameters

        # control parameters
        tau = 1 / np.sqrt(2 * np.sqrt(n))
        tau_prime = 1 / np.sqrt(2 * n)
        beta = 0.0873

        alpha_indices = [(i, j) for i in range(n - 1) for j in range(i + 1, n)]

        # assess initial population
        population_fvals = np.array([obj_func(x) for x in X])
        sorted_indices = np.argsort(population_fvals)
        best_x, best_fval = X[sorted_indices[0]], population_fvals[sorted_indices[0]]

        # save intermediate results for analysis
        history = {'X': [X], 'population_fvals': [population_fvals], 'SP': [SP],
                   'best_fval': [best_fval], 'convergence': -1}

        while n_fvals <= max_n_fvals:
            # select
            parents_indices = sorted_indices[:n_parents]
            X_parents = X[parents_indices]
            SP_parents = SP[parents_indices]
            fval_parents = population_fvals[parents_indices]

            if history['convergence'] == -1 and \
                    abs(min(fval_parents) - max(fval_parents)) < epsilon:
                # convergence criteria met
                history['convergence'] = n_fvals

            # recombine
            X = recombination(X_parents, method=x_recombination_method,
                              n_offspring=population)
            SP = recombination(SP_parents, method=sp_recombination_method,
                               n_offspring=population)

            # mutate
            X, SP = mutation(X, SP, tau, tau_prime, beta, alpha_indices,
                             mutate_angles=mutate_angles)

            # remove invalid solutions
            mask = np.all((X >= -2) & (X <= 2), axis=1)
            X = X[mask]
            SP = SP[mask]

            # evaluate
            population_fvals = np.array([obj_func(x) for x in X])

            if select_parents:      # mu + lambda selection scheme
                # include parents in new population
                X = np.concatenate((X, X_parents), axis=0)
                population_fvals = np.concatenate((population_fvals, fval_parents), axis=0)
                SP = np.concatenate((SP, SP_parents), axis=0)

            # sort population by fval for selection
            sorted_indices = np.argsort(population_fvals)
            if population_fvals[sorted_indices[0]] < best_fval:
                best_x, best_fval = \
                    X[sorted_indices[0]], population_fvals[sorted_indices[0]]

            # record for later analysis
            history['X'].append(X)
            history['population_fvals'].append(population_fvals)
            history['SP'].append(SP)
            history['best_fval'].append(best_fval)

    except MaxFuncEvaluationsExceeded:
        history['best_fval'] = best_fval_history.copy()
        history['timestamps'] = timestamps.copy()
        return best_x, best_fval, history

    history['best_fval'] = best_fval_history.copy()
    history['timestamps'] = timestamps.copy()
    return best_x, best_fval, history


def reset():
    global n_fvals, start_time
    n_fvals = 0
    start_time = time()
    best_fval_history.clear()
    timestamps.clear()


def shubert(x):
    a = np.arange(1, 6)
    A = np.tile(a, (x.shape[0], 1))
    fval = np.sum( np.dot( np.sin((A + 1) * np.expand_dims(x, -1) + A), a ) )
    return fval


def shubert_with_count(x, max_n_fvals):
    global n_fvals
    if n_fvals >= max_n_fvals:
        # raise exception to be caught by try block
        raise MaxFuncEvaluationsExceeded()

    fval = shubert(x)

    n_fvals += 1
    if not best_fval_history or fval < best_fval_history[-1]:
        best_fval_history.append(fval)
    else:
        best_fval_history.append(best_fval_history[-1])

    return fval


class MaxFuncEvaluationsExceeded(Exception):
    pass


if __name__ == '__main__':
    # example code
    n = 5
    max_n_fvals = 1e4
    obj_func = lambda x: shubert_with_count(x, max_n_fvals=max_n_fvals)

    best_x, best_fval, history = \
        evolution_strategy(obj_func, n, population=105, n_parents=15, s0=0.1,
                           x_recombination_method='discrete',
                           sp_recombination_method='global discrete',
                           select_parents=False, mutate_angles=False,
                           max_n_fvals=1e4)

    print(best_fval)
```

A.2 Tabu Search – Python Code

tabu_search.py

```python
import numpy as np
from collections import deque
from time import time

n_fvals = 0
best_fval_history = []
timestamps = []
start_time = 0


def update_stm(stm, x, stm_size):
    stm.append(tuple(x))
    if len(stm) > stm_size:
        stm.popleft()


def update_mtm(mtm, x, fval, mtm_size):
    x = tuple(x)
    if x not in mtm:
        if len(mtm) < mtm_size:
            mtm[x] = fval
        else:
            # replace the worst solution in MTM with x
            worst_x = max(mtm, key=mtm.get)
            if fval < mtm[worst_x]:
                mtm.pop(worst_x)
                mtm[x] = fval


def update_ltm(ltm, x, bounds, n_sectors):
    if np.any((x < bounds[0]) | (x > bounds[1])):
        # do not update ltm if out of bound
        return

    # subtract a small epsilon if any component = upper bound (for floor division)
    x[x == bounds[1]] = bounds[1] - 1e-10

    sector_length = (bounds[1] - bounds[0]) / n_sectors
    # calculate sector numbers of x
    sector = tuple((x - bounds[0]) // sector_length)
    if sector not in ltm:
        ltm[sector] = {tuple(x)}
    else:
        ltm[sector].add(tuple(x))


def local_search(x0, obj_func, bounds, step_size, stm):
    fval0 = obj_func(x0)
    best_x = None
    best_fval = fval0

    for i in range(x0.shape[0]):
        x = x0.copy()
        x[i] += step_size
        # check that location is not in stm and is valid
        if tuple(x) not in stm and np.all((x >= bounds[0]) & (x <= bounds[1])):
            tmp_fval = obj_func(x)
            if best_x is None or obj_func(x) < best_fval:
                best_x = x
                best_fval = tmp_fval

        # do the same thing for x_i - step_i
        x = x0.copy()
        x[i] -= step_size
        if tuple(x) not in stm and np.all((x >= bounds[0]) & (x <= bounds[1])):
            tmp_fval = obj_func(x)
            if best_x is None or obj_func(x) < best_fval:
                best_x = x
                best_fval = tmp_fval

    if best_x is None:
        # only happens when all solutions are rejected
        return x0, fval0

    # pattern move
    tmp_x = best_x + best_x - x0
    if tuple(tmp_x) not in stm and np.all((tmp_x >= bounds[0]) & (tmp_x <= bounds[1])):
        tmp_fval = obj_func(tmp_x)
        if best_fval < fval0 and tmp_fval < best_fval:
            best_x = tmp_x
            best_fval = tmp_fval

    return best_x, best_fval


def intensify(mtm, n):
    # return mean location in MTM
    X = np.array(list(mtm.keys()))
    return X.mean(axis=0)


def diversify(ltm, n, n_sectors, bounds, explored_thresh):
    if len(ltm) == n_sectors ** n and \
            all([len(locs) > explored_thresh for locs in ltm.values()]):
        # all areas have been reasonably searched; pick the one with the smallest count
        sector = np.array(min(ltm, key=lambda k: len(ltm[k])))

    else:
        # randomly select an unexplored sector
        sector = np.random.randint(n_sectors, size=n)
        while tuple(sector) in ltm and len(ltm[tuple(sector)]) > explored_thresh:
            # if it is explored already, select another sector
            sector = np.random.randint(n_sectors, size=n)

    # sample uniformly in sector
    sector_length = (bounds[1] - bounds[0]) / n_sectors
    x = bounds[0] + sector_length * sector + np.random.uniform(0, sector_length, size=n)

    return x


def tabu_search(x, obj_func, n, bounds=(-2, 2), step_size=0.1, n_sectors=10,
                si_thresh=10, sd_thresh=15, ssr_thresh=25,
                stm_size=3, mtm_size=4, ssr_factor=0.5, explored_thresh=10,
                epsilon=1e-9, max_n_fvals=1e4):
    try:
        reset()

        stm = deque()       # queue data structure
        mtm = {}    # <tuple> x: <float> fval
        ltm = {}    # <tuple> sector: <set> visited_locations

        best_x = x
        best_fval = obj_func(x)

        # save intermediate results for analysis
        history = {'x': [x], 'fval':[best_fval], 'si': [], 'sd': [], 'ssr': [],
                   'convergence': -1}

        update_stm(stm, best_x, stm_size)
        update_mtm(mtm, best_x, best_fval, mtm_size)
        update_ltm(ltm, best_x, bounds, n_sectors)

        counter = 0
        while n_fvals <= max_n_fvals:
            if history['convergence'] == -1 and step_size < epsilon:
                # convergence criteria met
                history['convergence'] = n_fvals

            x, fval = local_search(x, obj_func, bounds, step_size, stm)

            history['x'].append(x)
            history['fval'].append(fval)

            update_stm(stm, x, stm_size)
            update_mtm(mtm, x, fval, mtm_size)
            update_ltm(ltm, x, bounds, n_sectors)

            if fval < best_fval:     # new best solution found
                best_x, best_fval = x, fval
                counter = 0     # reset counter
                continue    # ignore the remaining code in this iteration

            if counter == si_thresh:
                history['si'].append(n_fvals)

                x = intensify(mtm, n)
                fval = obj_func(x)

                history['x'].append(x)
                history['fval'][-1] = fval

                update_stm(stm, x, stm_size)
                update_mtm(mtm, x, fval, mtm_size)
                update_ltm(ltm, x, bounds, n_sectors)

                if fval < best_fval:
                    best_x, best_fval = x, fval
                    counter = 0     # reset counter
                    continue

            elif counter == sd_thresh:
                history['sd'].append(n_fvals)

                x = diversify(ltm, n, n_sectors, bounds, explored_thresh=explored_thresh)
                fval = obj_func(x)

                history['x'].append(x)
                history['fval'][-1] = fval

                update_stm(stm, x, stm_size)
                update_mtm(mtm, x, fval, mtm_size)
                update_ltm(ltm, x, bounds, n_sectors)

                if fval < best_fval:
                    best_x, best_fval = x, fval
                    counter = 0     # reset counter
                    continue

            elif counter == ssr_thresh:
                history['ssr'].append(n_fvals)

                step_size *= ssr_factor     # reduce step size
                x = best_x      # move x to best location

                history['x'].append(x)
                history['fval'].append(fval)

                update_stm(stm, x, stm_size)
                # no need to update MTM and LTM since x is already in them
```

```python

                counter = 0     # reset counter
                continue

            counter += 1

    except MaxFuncEvaluationsExceeded:
        history['best_fval'] = best_fval_history.copy()
        history['timestamps'] = timestamps.copy()
        return best_x, best_fval, history

    history['best_fval'] = best_fval_history.copy()
    history['timestamps'] = timestamps.copy()
    return best_x, best_fval, history


def reset():
    global n_fvals, start_time
    n_fvals = 0
    start_time = time()
    best_fval_history.clear()
    timestamps.clear()


def shubert(x):
    a = np.arange(1, 6)
    A = np.tile(a, (x.shape[0], 1))
    fval = np.sum( np.dot( np.sin((A + 1) * np.expand_dims(x, -1) + A), a ) )
    return fval


def shubert_with_count(x, max_n_fvals):
    global n_fvals
    if n_fvals >= max_n_fvals:
        # raise exception to be caught by try block
        raise MaxFuncEvaluationsExceeded()

    fval = shubert(x)

    n_fvals += 1
    if not best_fval_history or fval < best_fval_history[-1]:
        best_fval_history.append(fval)
    else:
        best_fval_history.append(best_fval_history[-1])

    return fval


class MaxFuncEvaluationsExceeded(Exception):
    pass


if __name__ == '__main__':
    # example code
    n = 5
    x = np.random.uniform(-2, 2, size=n)
    max_n_fvals = 1e4
    obj_func = lambda x: shubert_with_count(x, max_n_fvals)

    best_x, best_fval, history = \
        tabu_search(x, obj_func, n, step_size=1, n_sectors=5,
                si_thresh=10, sd_thresh=15, ssr_thresh=25,
                stm_size=26, mtm_size=4, ssr_factor=0.8, explored_thresh=10,
                epsilon=1e-9, max_n_fvals=max_n_fvals)

    print(best_fval)
```