Solving Optimisation Problems

Issei Kuzuki

1 Introduction

1.1 Task

The task covered in this report is to attempt to minimise the 6-dimensional Schwefel's Function using 2 different optimisation algorithms.

$$Minimise \qquad f(x) = \sum_{i=1}^{6} \left[-x_i sin(\sqrt{|x_i|})\right]$$

$$Subject\ to \qquad -500 \leq x_i \leq 500$$

The Schwefel's function has multiple local maxima and minima and therefore is a complicated optimisation problem to solve.

The only restriction is that each run may have a maximum of 15,000 objective function evaluations.

1.2 Simulated annealing

Simulated annealing is an optimisation algorithm that uses a probabilistic technique to approximate the global optimal solution of a given problem by mimicking the process of physical annealing. The algorithm starts with an initial solution and iteratively makes small changes to the solution in order to improve it. The probability of accepting a new solution that is worse than the current one is determined by a probability distribution called a Boltzmann distribution. The algorithm gradually reduces the temperature, which controls the probability of accepting a worse solution, until it reaches a very low temperature where only the best solutions are accepted. This allows the algorithm to escape local optima and converge to a global optimum.

The way this particular algorithm has been implemented to deal with constraints is that in each iteration, a step is taken for each of the variables, the step taken is random and is determined by a normal distribution. The algorithm then checks whether the variable is within the allowed constraints, if it is not, new steps are taken until they are.

The temperature is decreased after a set number of trials L_k or a minimum number of acceptances η_{min}, whichever comes first. For this algorithm $\eta_{min} = 0.6L_k$ was used as suggested in the lecture notes.

Figure 1 shows an example of the progress of the simulated annealing algorithm on the 2D version of the Schwefel's function.

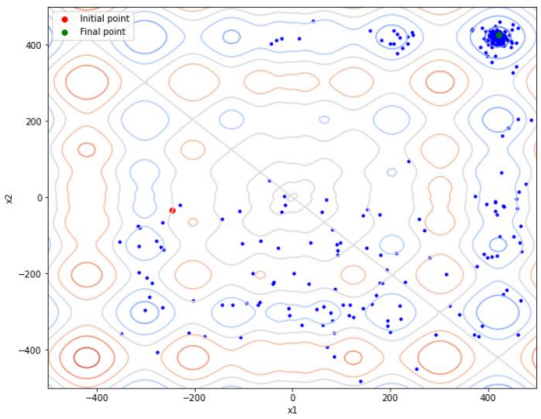


Figure 1: Contour plot of 2D Schwefel's function being solved by simulated annealing

1.3 Genetic algorithm

Genetic algorithm uses the principles of natural selection to evolve the population towards better solutions. Initially, a population of points is selected and the fittest or best solutions have the highest probability of being selected as parents and passing on their genetic information to the next generation. The genetic operations of crossover and mutation introduce diversity into the population, crossover combines the genetic information of the selected parents to create a new offspring while mutation introduces small random changes to the genetic information of the offspring. These strategies help to escape local optima and explore the solution space.

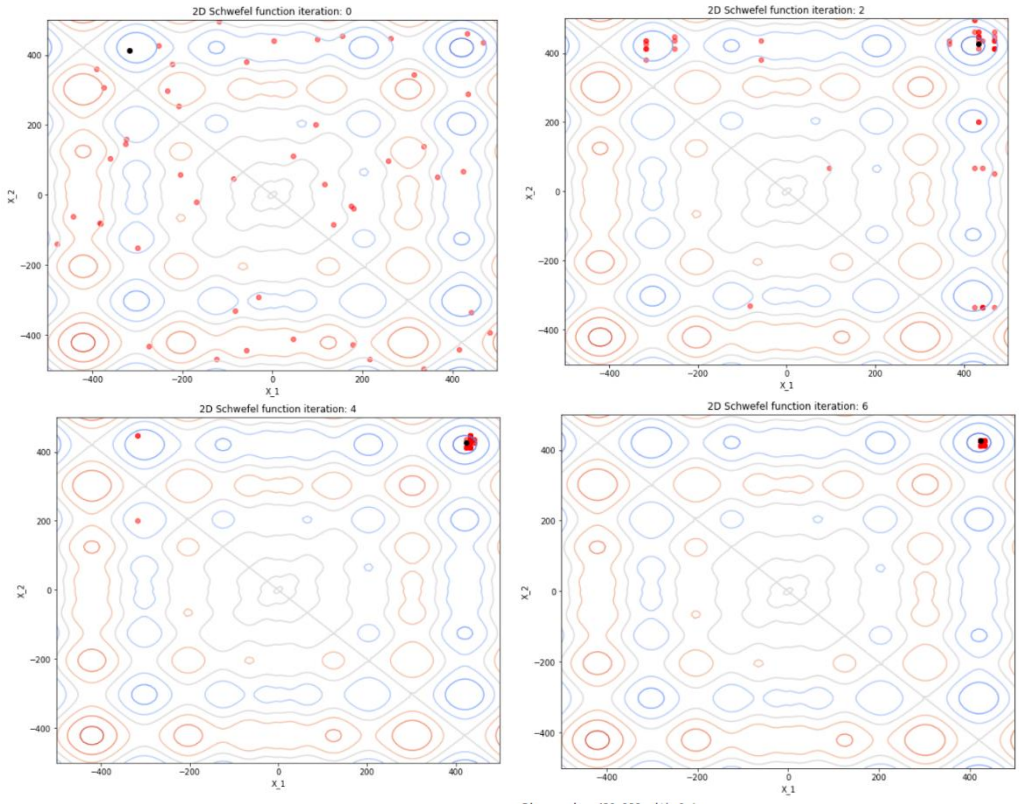


Figure 2: Contour plot of 2D Schwefel's function being solved by genetic algorithm

Figure 2 shows an example of a genetic algorithm progressing on the 2D version of the Schwefel's function. The red dots represent the population of 50 points at each iteration while the black point shows the 'fittest' solution in the current population.

2 Simulated annealing experiments

In order to optimise the performance of the simulated annealing algorithm, several experiments were performed varying the parameters of the algorithm to see how it affected the mean and standard deviation of objective function values. For each set of parameters, the algorithm was run 50 times using the same set of randomly chosen initial points.

2.1 Step size

The step size refers to the magnitude by which the normal distribution is multiplied to decide the random perturbation of the variables at each iteration.

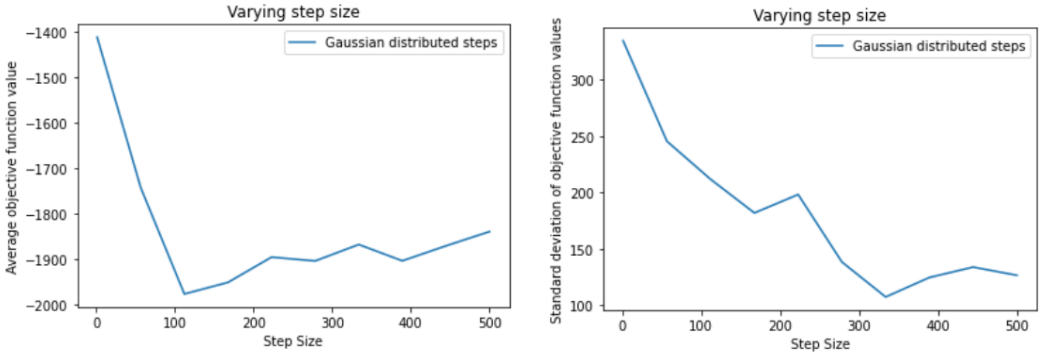


Figure 3: Performance of algorithm with varying step sizes

From figure 3, it can be seen that the mean of the objective function value reaches a optimum at a step size of around 105 while the performance deteriorates for larger and smaller step sizes. The standard deviation of the objective function value shows a general decreasing trend with increasing step size.

This is because a small step size will allow the algorithm to explore the solution space slowly increasing the chances of finding the global optimal solution. With a small step size, the algorithm makes only small changes to the current solution at each step, but it also decreases the probability of accepting a worse solution. When the step size is too small, the number of iterations of the algorithm is not sufficient to find a suitable minima resulting in poor performance.

Meanwhile a large step size will allow the algorithm to explore the solution space more quickly but also increases the chances of getting stuck in local optima. With a large step size, the algorithm can move far from the current solution in a single step, which allows it to consistently find a decent local minima resulting in a lower standard deviation.

2.2 Initial temperature

The initial temperature effects the chance of solutions that increase the objective function being accepted. First an initial test was carried out where all increases in f are accepted, the average objective function increase observed $\overline{\delta f}^+$ and the standard deviation of the objective function values observed σ_0 were calculated. Three different initial temperatures were tested, the Kirkpatrick method:

$$T_0 = -\frac{\overline{\delta f}^+}{ln(0.8)}$$

Where 0.8 refers to the average probability of a solution that increases f being accepted.

The White formulation:

$$T_0 = \sigma_0$$

And a reference where an initial temperature of 1 was selected.

	T_0	$\overline{f(x)}$	$\sigma_{f(x)}$
Reference	1	-1980.4	188.3
Kirkpatrick	2135.2	-1983.8	197.3
White	600.9	-2026.4	185.3

Table 1: Performance of algorithm with different initial temperatures

The choice of the initial temperature is a trade-off between the exploration of the solution space and the likelihood of finding the global optimal solution, a high initial temperature will increase the probability of accepting a worse solution, which allows the algorithm to explore the solution space more extensively. With a high initial temperature, the algorithm can move far from the current solution in a single step, but it also increases the chances of getting stuck in local optima.

On the other hand, a low initial temperature will decrease the probability of accepting a worse solution, which allows the algorithm to converge faster to the global optimal solution. With a low initial temperature, the algorithm makes only small changes to the current solution at each step, but it also decreases the chances of finding the global optimal solution.

Table 1 shows the resulting mean and standard deviation of the objective function values found using the three initial temperatures. The White formulation gives the best performance overall.

2.3 Markov chain length

The Markov chain length refers to the number of iterations of the algorithm needed before the temperature is dropped. As discussed in 1.2, the drop can be triggered by a set number of trials L_k or a minimum number of acceptances η_{min}. A fixed ratio of $\eta_{min} = 0.6L_k$ was used so the Markov chain length was varied by changing the minimum number of trials needed to trigger a drop in temperature.

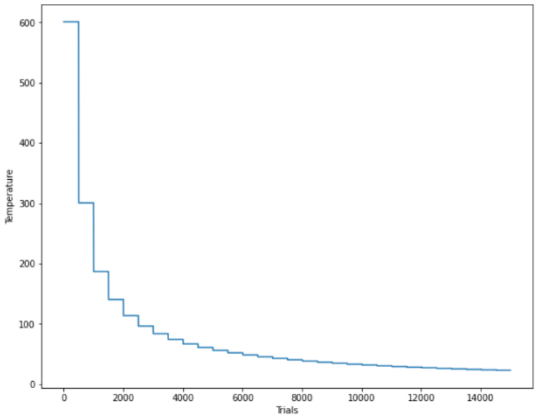


Figure 4: Example annealing schedule

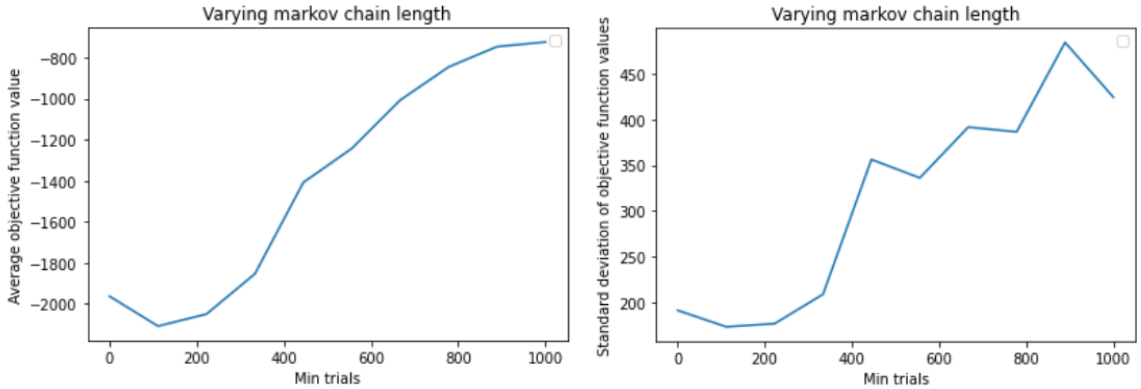


Figure 5: Performance of algorithm with varying η_{min}

A large L_k to decrement the temperature will reduce the temperature less frequently, which will allow the algorithm to explore the solution space more extensively before trying to converge to a global optimum. This can help the algorithm to find the global optimum, but it also increases the chances of getting stuck in local optima.

On the other hand, a small L_k will reduce the temperature more frequently, which will cause the algorithm to converge faster to the global optimum. This can help the algorithm to avoid getting stuck in local optima, but it also decreases the chances of finding the global optimal solution.

The choice of the initial temperature is a trade-off between the exploration of the solution space and the likelihood of finding the global optimal solution, from figure 5 it can be seen that the optimal performance for this particular problem occurs at around $L_k = 100$ giving $\eta_{min} = 60$.

2.4 Temperature decrement rule

The standard temperature decrement rule is an exponential cooling scheme:

$$T_{k+1} = \alpha T_k$$

Where α is a fixed value, the ratio between the old and new temperatures.

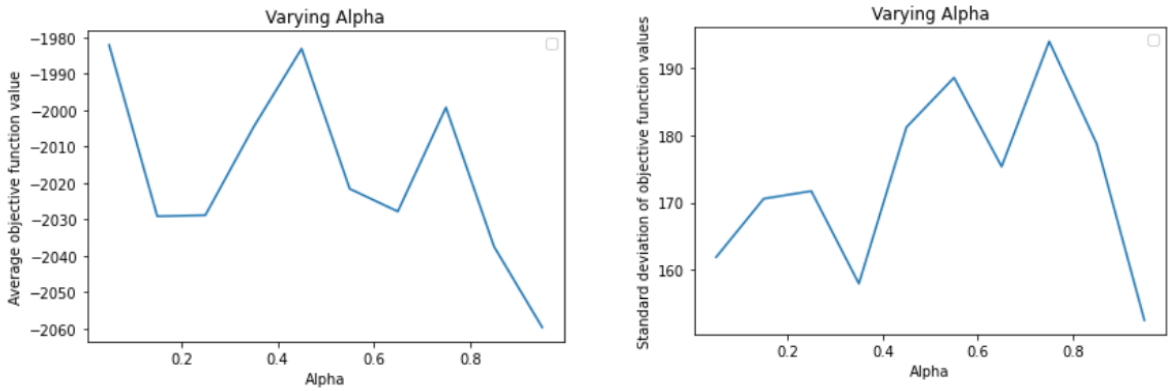


Figure 6: Performance of algorithm with varying Alpha

A low value for alpha will result in a rapid decrease in temperature, which will cause the optimisation algorithm to quickly converge on a global optimum. However, this could also lower the chances of finding the global optimal solution as the algorithm may not explore the solution space enough before converging.

On the other hand, a high value for alpha will result in a slower decrease in temperature, which will allow the algorithm to explore the solution space more extensively before trying to converge on a global optimum. This can increase the chances of finding the global optimal solution, but it also increases the risk of getting stuck in local optima.

For this particular problem, there is no clear trend for how alpha effects the mean and standard deviation of the objective function however, the optimal value is found at alpha = 0.95.

An adaptive scheme uses the algorithms current performance to modify the value of alpha. The Huang formulation is:

$$T_{k+1} = \alpha_k T_k$$

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

Where σ_k is the standard deviation of the objective function values accepted at temperature T_k.

	$\overline{f(x)}$	$\sigma_{f(x)}$
Exponential cooling (α=0.95)	-2072.8	184.8
Adaptive cooling	-2039.9	151.0

Table 2: Performance of algorithm with different cooling schemes

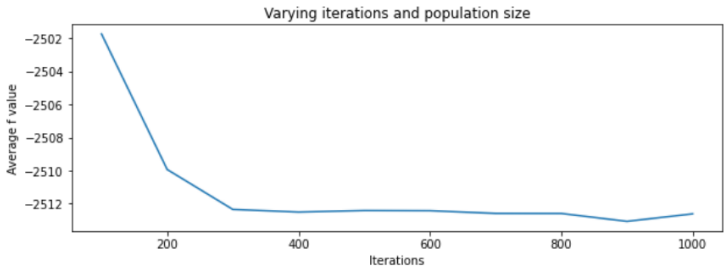
In theory, an adaptive scheme where the optimisation starts with a high alpha value and gradually decrease over time should allow the early stages of the optimisation to explore the solution space quickly, and the lower alpha value in the later stages to refine the solution and avoid getting stuck in a local optimum. However, from table 2 it is not clear whether the adaptive scheme has outperformed the exponential scheme. This could be due to difficulties in detecting convergence, an adaptive cooling schedule relies on detecting convergence to adjust the temperature but detecting convergence can be difficult, especially for a problem like the 6D Schwefel's function with many local optima and a complex solution space.

3 Genetic algorithm experiments

In order to optimise the performance of the genetic algorithm, several experiments were performed varying the parameters of the algorithm to see how it affected the mean and standard deviation of objective function values. Additionally, the average run time for the algorithm was measured. For each set of parameters, the algorithm was run 25 times.

3.1 Iterations and population size

Iterations refer to the number of times a new population is created while the population size is the quantity of members in the population at each iteration. The product of these quantities defines the number of objective function evaluations. Since the number of objective function evaluations must remain fixed at 15,000, the number of iterations was varied together with the population size.



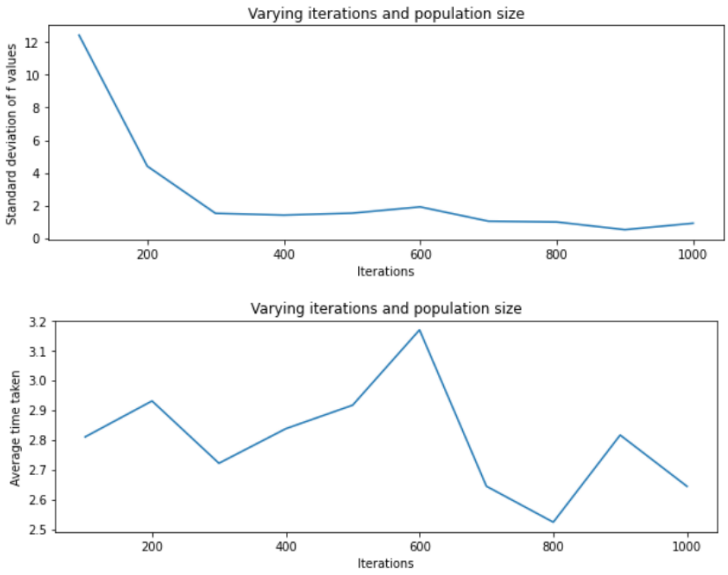


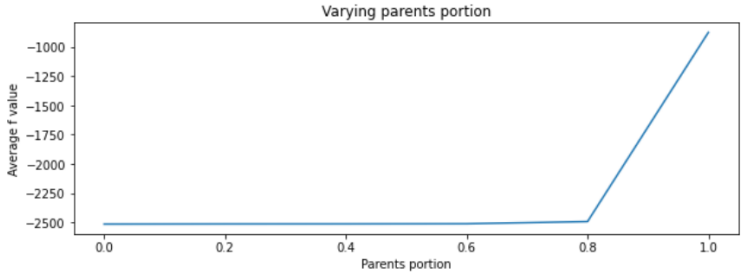
Figure 7: Performance of algorithm with varying iterations and population size

From figure 7 it can be seen that the performance of the algorithm improves with increasing the number of iterations and consequently reducing the population size. In general, increasing the number of iterations allows the algorithm to run for a longer time, which can increase the chances of finding the global optimum. But increasing the population size also allows for a larger number of individuals to be evaluated and evolved over the course of the optimisation, which can increase the chances of finding the global optimum.

The explanation for the trends in figure 7 is likely due to a larger population size having diminishing returns for the performance. There is no clear trend in the average computational time for the algorithm with increasing iterations, this is because the number of objective function evaluations remains constant.

3.2 Parents portion

The parents portion is the portion of the new population filled by the members of the previous population.



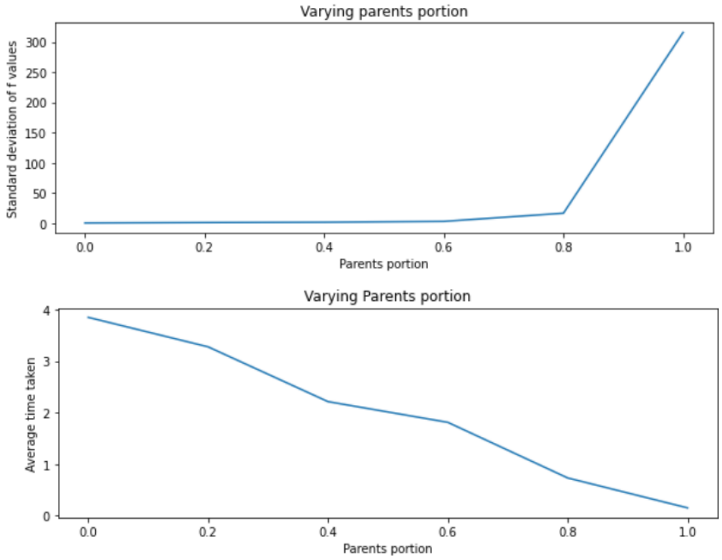


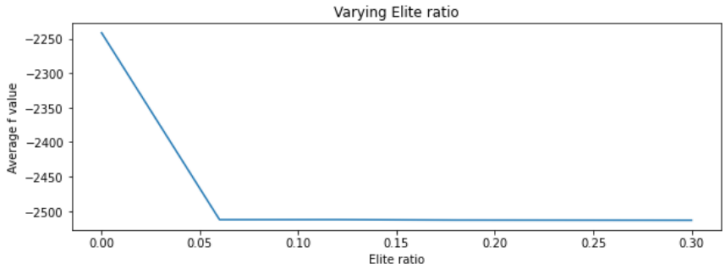
Figure 8: Performance of algorithm with varying parents portion

Increasing the parents portion will lead to a higher proportion of the population being selected as parents, which can increase the chances of good solutions being passed on to the next generation. However, it also increases the risk of losing diversity in the population, as fewer individuals are selected for mutation.

For this particular problem, figure 8 shows that a low parents portion is desirable for the objective function value but there is a trade-off with the average computational time since a low parents portion means more new individuals must be created at each iteration. A good compromise would be a parents portion of around 0.4.

3.3 Elite ratio

The elite ratio is a parameter that controls the proportion of the population that is selected to be carried over to the next generation without undergoing genetic operations. This group of individuals, known as the "elite", is considered to be the best solutions found so far and are preserved to ensure that good solutions are not lost during the optimisation process. By definition, the elite ratio must be less than the parents portion.



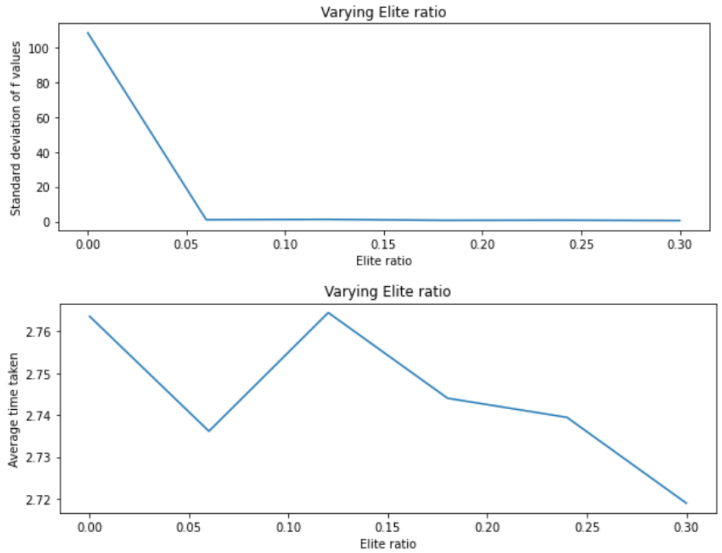


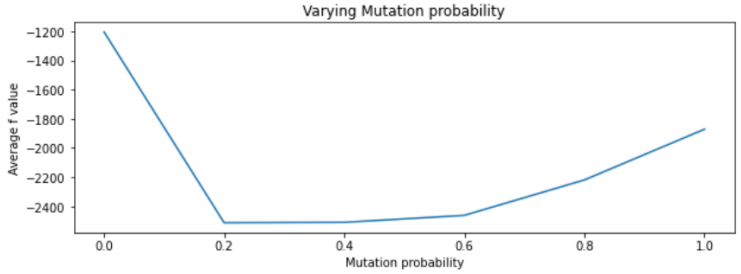
Figure 9: Performance of algorithm with varying elite ratio

The elite ratio can play an important role in the performance of a genetic algorithm. A high elite ratio can help to preserve good solutions and prevent the algorithm from getting stuck in local optima, while a low elite ratio can allow the algorithm to explore the solution space more extensively and improve its chances of finding the global optimum. However, a too high elite ratio can also lead to premature convergence, as the diversity of the population is reduced.

In the range experimented over, a high elite ratio resulted in optimal performance with 0.3 resulting in the lowest average objective function value and standard deviation.

3.4 Mutation probability

The mutation probability is a parameter that controls the likelihood of a genetic mutation occurring during the genetic operation of mutation. Mutation is a random process that introduces small variations into the genetic information of an individual, and it is used to maintain diversity in the population and to explore new regions of the solution space.



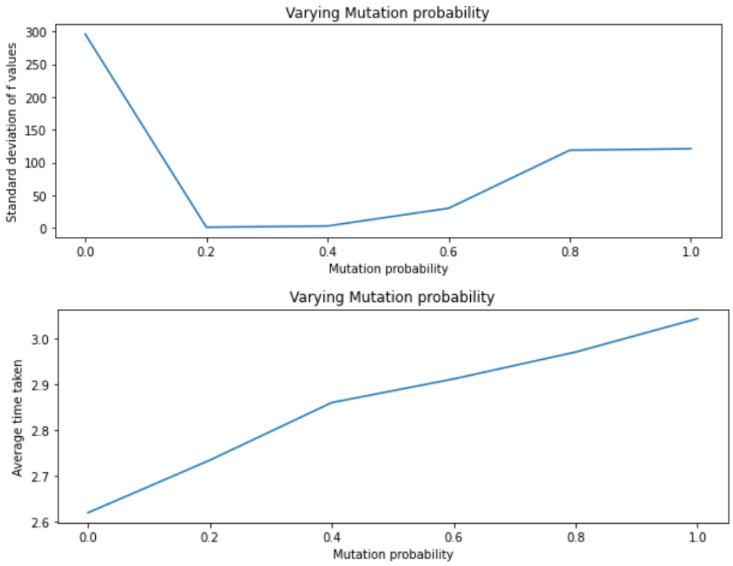


Figure 10: Performance of algorithm with varying mutation probability

A high mutation probability can help to maintain diversity in the population and improve the chances of finding the global optimum, while a low mutation probability can help to preserve good solutions and prevent the algorithm from getting stuck in local optima. However, a too high mutation probability can also lead to a loss of good solutions, as well as a loss of the direction towards the global optimum.

This trade-off is evident in figure 10 and the optimal performance of the algorithm occurs at a mutation probability of 0.2.

3.5 Crossover type

The crossover type is a parameter that controls how the genetic information of two parent individuals is combined to create a new offspring. Crossover is a genetic operator used to combine the characteristics of multiple solutions and to create new solutions with a better chance of solving the problem. The crossover type determines the specific method used to combine the genetic information of the parents.

The three methods explored here are:

One-point crossover: a single position on the 6D vector is chosen at random, and the elements to the left of that point is inherited from one parent, while the elements to the right is inherited from the other.

Two-point crossover: two positions on the 6D vector are chosen at random, and the elements between those points are inherited from one parent while the remaining is inherited from the other.

Uniform crossover: A random number is generated for each element of the vector and if the number is below a certain threshold, element is inherited from one parent, otherwise it is inherited from the other.

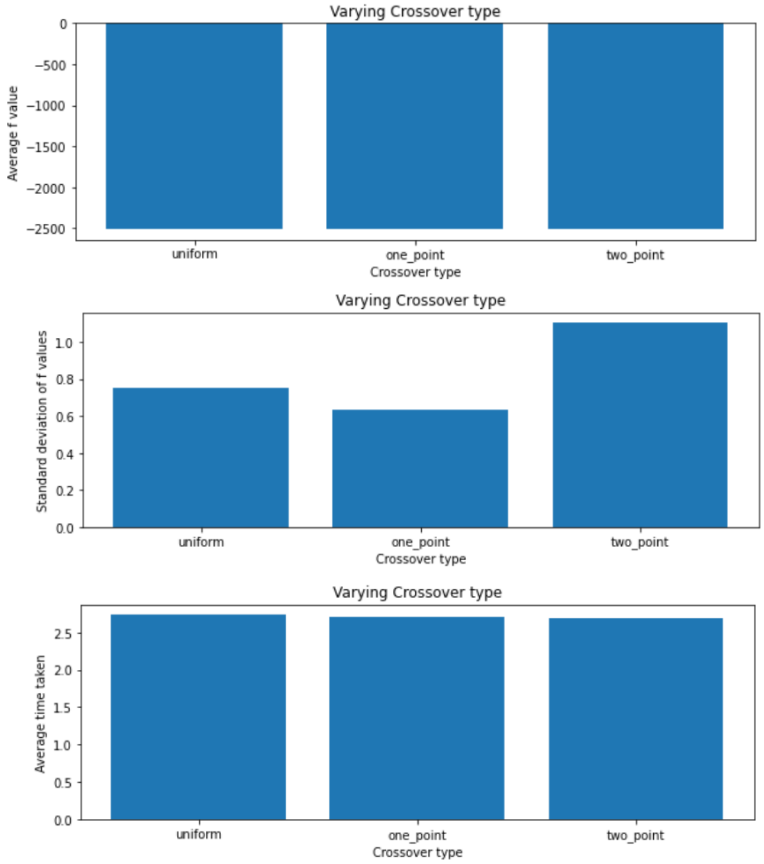


Figure 11: Performance of algorithm with varying crossover type

One-point or two-point crossover are more likely to retain good solutions and prevent the algorithm from getting stuck in local optima, while other crossover types like uniform crossover or arithmetic crossover are more likely to explore the solution space and improve the chances of finding the global optimum.

Figure 11 shows that for this problem, the crossover types have little effect on the algorithm's performance except for a small difference in the standard deviation of the objective function values obtained.

3.6 Crossover probability

The crossover probability is typically a value between 0 and 1, and it is applied to each pair of parent individuals. When two parents are selected for crossover, a random number is generated between 0 and 1, and if the number is below the crossover probability, the

parents will undergo crossover to create offspring. If the number is above the crossover probability, the parents will be copied to the next generation without undergoing crossover.

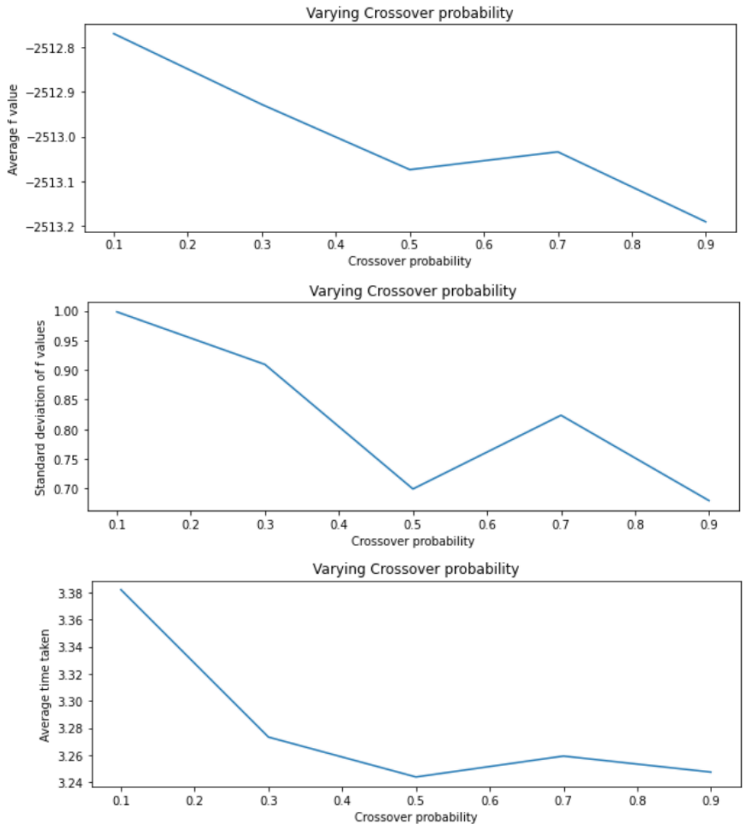


Figure 12: Performance of algorithm with varying crossover probability

A high crossover probability can help to maintain diversity in the population and improve the chances of finding the global optimum, while a low crossover probability can help to preserve good solutions and prevent the algorithm from getting stuck in local optima. However, a too high crossover probability can also lead to a loss of good solutions, as well as a loss of the direction towards the global optimum.

From figure 12, it can be seen that for this problem, the performance of the algorithm becomes better with crossover probability as well as decreasing the average computational time.

4 Comparison of methods

4.1 Performance

Following the experiments in section 3, the algorithms were run 50 times using their best parameters and the performance comparison can be found in table 3:

	$\overline{f(x)}$	$\sigma_{f(x)}$	Average run time
Simulated annealing	-2057.7	182.3	0.884
Genetic algorithm	-2513.0	0.9	3.798

Table 3: Performance of optimised algorithms

4.2 Evaluation

From table 3, it is evident that the genetic algorithm has significantly outperformed the simulated annealing method, consistently finding a better optimal value.

A possible explanation for this is that the genetic algorithm is well-suited for problems with complex and large search spaces like the 6D Schwefel's function in this problem. Moreover, it can handle problems with multiple local minima, because it uses a population of solutions instead of a single solution like the simulated annealing method.

Additionally, the simulated annealing algorithm being used was a rather simple version, this is evident from the average run times of the algorithms. A more complex approach which starts with a large step size and then gradually reduces it as the search progresses could result in better performance, this is because the larger step size can help to explore the solution space quickly in the early stages of the optimisation, and the smaller step size in the later stages can help to refine the solution and avoid getting stuck in a local optimum.

5 Appendix

5.1 Simulated annealing implementation code

```
import numpy as np
#Define Schwefel's function
def schwefel(x):
    return -sum(x_i * np.sin(np.sqrt(abs(x_i))) for x_i in x)
#Random initial solutions
import random

x0_list = [[random.randint(-
500, 500) for i in range(6)] for j in range(50)]
print(x0_list)

# maximum number of iterations
max_iter = 15000
# Function bounds
bounds = [-500, 500]
#List of step sizes
```

```
#Function to run simulated annealing algorithm
def SA_final(func, x0, T0, max_iter, bounds, step_size, min_trials, Alp
ha):
  # Start simulated annealing
  # initialise parameters
  x = x0
  T=T0
  counter = 0
  trial_counter = 0
  acceptance_counter = 0
  min_acceptance = 0.6*min_trials
  current_func = func(x)


  while counter < max_iter:
      # generate a random neighbor
      x_new = x + np.random.normal(size=6) * step_size
      # check if the new solution is within the bounds
      if any(x_i < bounds[0] or x_i > bounds[1] for x_i in x_new):
          continue
      # compute the energy change
      new_func = func(x_new)
      delta_E = new_func - current_func
      # decide whether to accept the new solution
      if delta_E < 0 or np.random.rand() < np.exp(-delta_E / T):
          x = x_new
          current_func = new_func
          acceptance_counter +=1
      # decrease the temperature
      counter += 1
      trial_counter +=1

      if trial_counter > min_trials or acceptance_counter > min_accepta
nce:
        T = T*Alpha
        trial_counter = 0
        acceptance_counter = 0


  return x, func(x)


#Performance test
import time
start_time = time.time()
objfunc = []
```

```
for i in range(len(x0_list)):
    objfunc.append(SA_final(schwefel, x0_list[i], T0_White, max_iter, b
ounds, 105, 105, 0.95)[1])

end_time = time.time()

avg_value = np.mean(objfunc)
std_value = np.std(objfunc)

time_taken = (end_time - start_time)/len(x0_list)

print(avg_value_original, std_value_original, time_taken_orig)
```

5.2 Genetic algorithm implementation code

The Python library 'geneticalgorithm' was used to implement this algorithm

Source: https://pypi.org/project/geneticalgorithm/

```
!pip install geneticalgorithm

import numpy as np
from geneticalgorithm import geneticalgorithm as ga

#Define Schwefel's function
def schwefel(x):
    return -sum(x_i * np.sin(np.sqrt(abs(x_i))) for x_i in x)

#Setting bounds
varbound=np.array([[-500,500]]*6)

#Algorithm parameters for the GA model

algorithm_param = {'max_num_iteration': 150,\
                   'population_size':100,\
                   'mutation_probability':0.1,\
                   'elit_ratio': 0.01,\
                   'crossover_probability': 0.5,\
                   'parents_portion': 0.3,\
                   'crossover_type':'uniform',\
                   'max_iteration_without_improv':None}

#Initial run of model

model=ga(function=schwefel,dimension=6,variable_type='real',variable_bo
undaries=varbound, algorithm_parameters=algorithm_param)

model.run()
```

```
#Define a function which runs the genetic algorithm 25 times: returns
the average value, standard deviation of the objective function as well
as the average computational time for each run

def test_run(max_num_iteration, population_size, mutation_probability,
elit_ratio, crossover_probability, parents_portion, crossover_type, max
_iteration_without_improv):
  objfunc = []
  time_list = []
  for j in range(25):

    start_time = time.time()

    varbound=np.array([[-500,500]]*6)

    algorithm_param = {'max_num_iteration': max_num_iteration,\
                   'population_size':population_size,\
                   'mutation_probability':mutation_probability,\
                   'elit_ratio': elit_ratio,\
                   'crossover_probability': crossover_probability,\
                   'parents_portion': parents_portion,\
                   'crossover_type':crossover_type,\
                   'max_iteration_without_improv':max_iteration_without
_improv}

    model=ga(function=schwefel,dimension=6,variable_type='real',variabl
e_boundaries=varbound, algorithm_parameters=algorithm_param, convergenc
e_curve=False, progress_bar=False)

    model.run()

    objfunc.append(model.output_dict['function'])

    end_time = time.time()

    time_taken = end_time - start_time
    time_list.append(time_taken)

  avg_value = np.mean(objfunc)
  std_value = np.std(objfunc)
  avg_time = np.mean(time_list)

  return avg_value, std_value, avg_time
```

```
#Performance test
ave, std, tim = test_run(max_num_iteration = 1000, population_size = 15
, mutation_probability = 0.2, elit_ratio = 0.2,
           crossover_probability = 0.9, parents_portion = 0.3, crossove
r_type = 'uniform', max_iteration_without_improv = None)

print(ave, std, tim)
```