# 4M17: Practical Optimization
# Assignment 2

**Student Candidate Number: 5490C**

Jan 19, 2020

## 1. Introduction

In this report, the algorithmic performance of Simulated Annealing (SA) and Genetic Algorithm (GA) are compared, in attempting to solve the 5-dimensional Rana's Function (5D-RF). The n-dimensional constrained problem is defined as:

$$f(\boldsymbol{x}) = \sum_{i=1}^{n-1} \cos\left(\sqrt{|x_{i+1} + x_i + 1|}\right) \sin\left(\sqrt{|x_{i+1} - x_i + 1|}\right) + (1 + x_{i+1}) \cos\left(\sqrt{|x_{i+1} - x_i + 1|}\right) \sin\left(\sqrt{|x_{i+1} + x_i + 1|}\right)$$

Subject to:

$$x_i \in [-500, 500] \; for \; i = 1, \dots, n$$

We consider the 2-dimensional form of the problem for visualization of the search paths followed by the optimization methods, and to ensure the algorithms are performing as expected. The 3D plot and contour plots of the 2D-RF drawn are shown below.

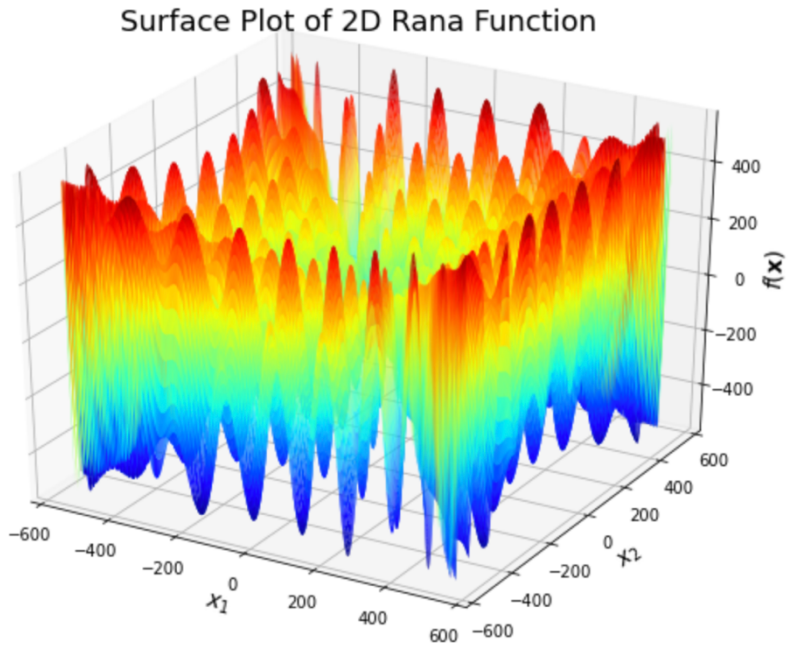


Figure 1. 3D plot of Rana's function for n=2

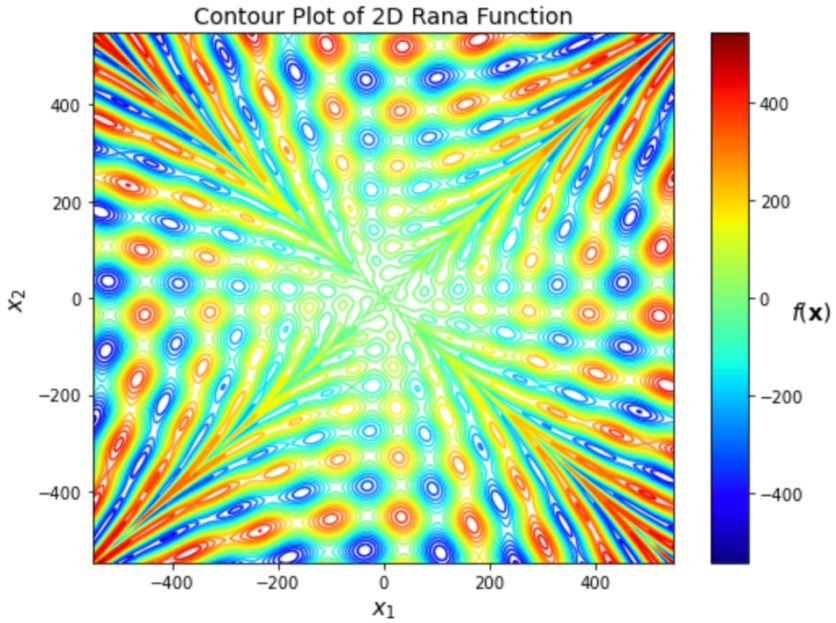


Figure 2. Contour plot of Rana's Function for n=2

## 2. Simulated Annealing

SA exploits the way in which a metal cools and freezes into a minimum energy structure. Its advantage over other methods is its ability to accept changes that increase the objective function. First, the algorithm implementation details are explained, with parameter changes made for investigation. Then, SA on 2D-RF was performed to visualize the algorithm, then the effect of parameter changes on SA's performance on 5D-RF was investigated. The algorithm was written in python Jupyter notebooks.

### 2.1 Algorithm description and implementation details

As our problem of minimizing Rana's function has continuous random variables, methods of generating new trial solutions that were used were in the form:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + D\boldsymbol{u}$$

Where $\boldsymbol{u}$ is a vector of random uniform numbers in the range (-1,1) and $D$ is a diagonal matrix. The strategy suggested by Parks [1990] in the lecture notes [1] was used.

To implement this, control variables are rescaled so that within the optimization routine they vary over the range (0,1),

According to the annealing schedule, standard implementation of the algorithm in which Markov chains are generated at decreasing temperatures was used.

The initial temperature is determined via one of 2 methods:

- Method by Kirkpatrick [1984], as mentioned in the lecture notes [1].
- Method by White [1984], as mentioned in the lecture notes [2].

The final temperature is determined by fixing the number of temperature values to be used, '$k$'. and each Markov chain is terminated after '$L_k$' trials. Different combinations of $(k, L_k)$ were investigated in the 5D-RF optimization.

Moreover, temperature is decremented according to one of two methods:

- The exponential cooling scheme (ECS) proposed by Kirkpatrick et al. [1982], from lecture notes [1]
- Method by Haung et al. [1986], from lecture notes [1].

### 2.2 2D-RF Simulated Annealing

For both the 2D-RF SA and 5D-RF SA investigations, the parameters in Table 1 below were used as the control, unless stated in each section, where changes in the parameters are to be investigated.

| Parameter | Value |
|---|---|
| $L_k$ | 300 |
| $k$ | 30 |
| $L_{k\,init}$ | 100 |
| Initial Temperature Scheme | White [1984] |
| Temperature decrement Scheme | Kirkpatrick [1982], $\alpha = 0.95$ |

Table 1. Standard Parameters used for SA (Control)

One particular run of the SA algorithm using the control parameters is visualized below.

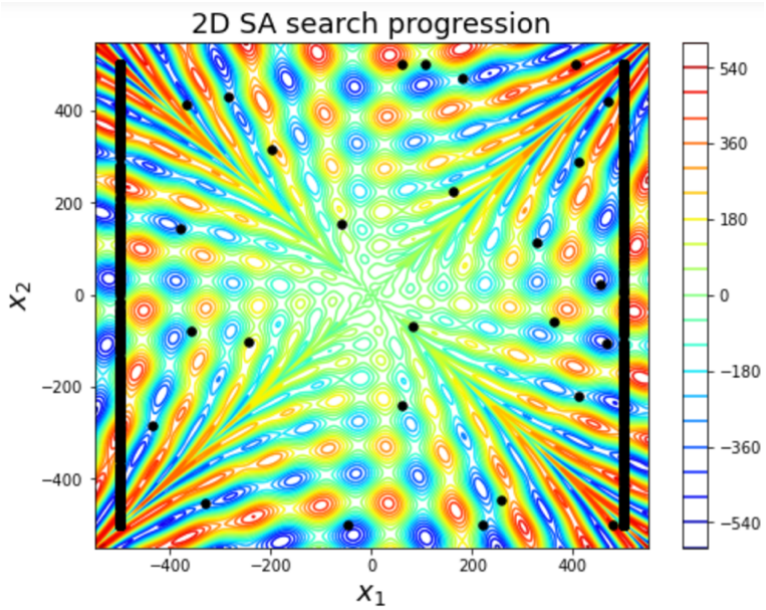


Figure 3. Search pattern of a particular search plotted on contour map for SA on 2DRF

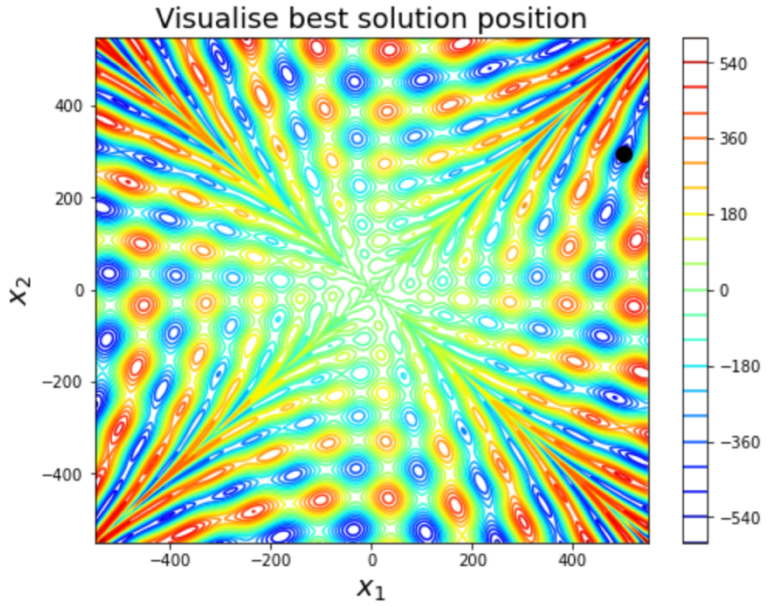


Figure 4. Location of global minimum for SA on 2DRF (top right)

The final minimum may not be the global minimum, as the search was only performed once, but it can be observed from figure 3 that the search performs as expected, exploring a large proportion (if not all) of the search space. For the particular search in Figure 3, where the SA arrived at the best minimum point of $f(\boldsymbol{x}) = -497.7306$ and $(x_1, x_2) = (500.0, 295.8754)$.

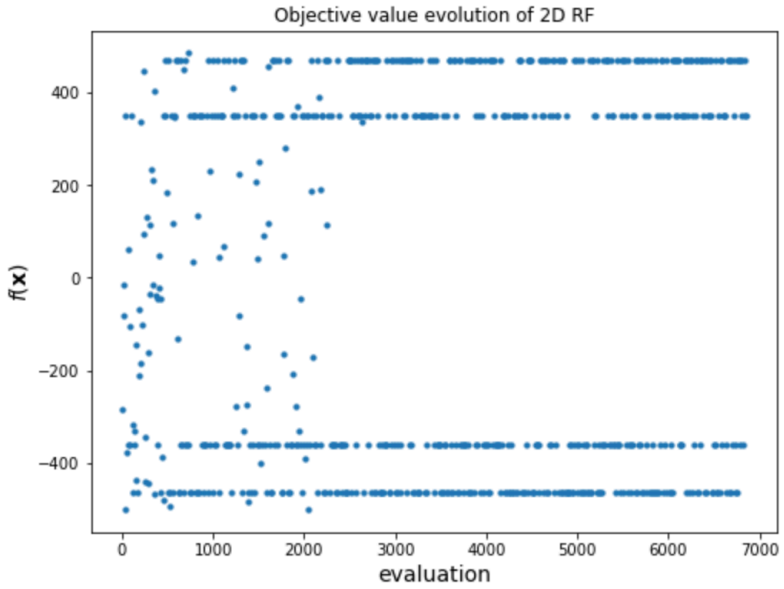


Figure 5. Objective values during 2D-RF SA search progression for a particular search

As it can be deduced from Figure 5, the objective function values stabilize at about 2000 iterations, resulting to jumping between 4 points as the search progresses. This is as expected, because of the natural complexity of the function as it can be seen from the heat map, Figure 2, and the way the function continuously alternates between a maximum and minimum in small changes in $x$. Moreover, SA is susceptible to accepting positive changes in the objective function, which explains why the value jumps between positive and negative values. The symmetric nature of the Rana function in $cos$ and $sine$ also explain the symmetry of the 4 alternating minimum points. This result suggests that for the Rana function, it may be highly difficult and computationally expensive to arrive at the global optimum. In the future, optimization could be simplified and narrowed down by initializing the searches close to these seen 4 minimum points.
Using the same parameters, search was iterated 50 times with different random seeds for reliability, and the histogram of resulting minimized objective function values is drawn.

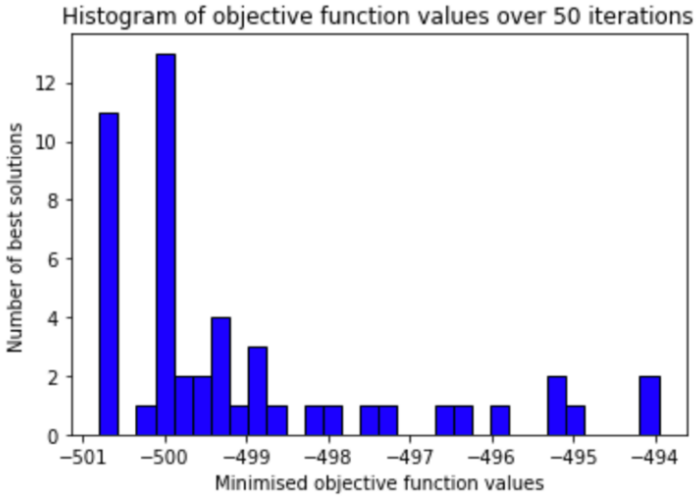


Figure 6. Histogram of the minimized objective function values over 50 iterations for SA 2D-RF

The histogram shows that the algorithm obtains a solution in a similar region (-500 to -501) and performs rather well.

### 2.3 5D-RF Simulated Annealing.

For the 5D SA algorithm, 3 parameters were investigated to explore the effects they have on algorithmic performance: different initial temperature determination schemes, different temperature decrement schemes, and different combinations of $(k, L_k)$. For each investigation, the algorithms were repeated 50 times.

#### 2.3.1 Initial Temperature (Kirkpatrick [1984] vs White [1984])

| | $f_{min}$ best | $f_{min}$ average | $\sigma_f, (s.d\ of\ f_{min})$ |
|---|---|---|---|
| Kirkpatrick | -1929.00 | -1870.94 | 23.57 |
| White, Control | -1953.22 | -1875.59 | 27.21 |

Table 2. Effect of 2 different initial temperature schemes on 5D-RF

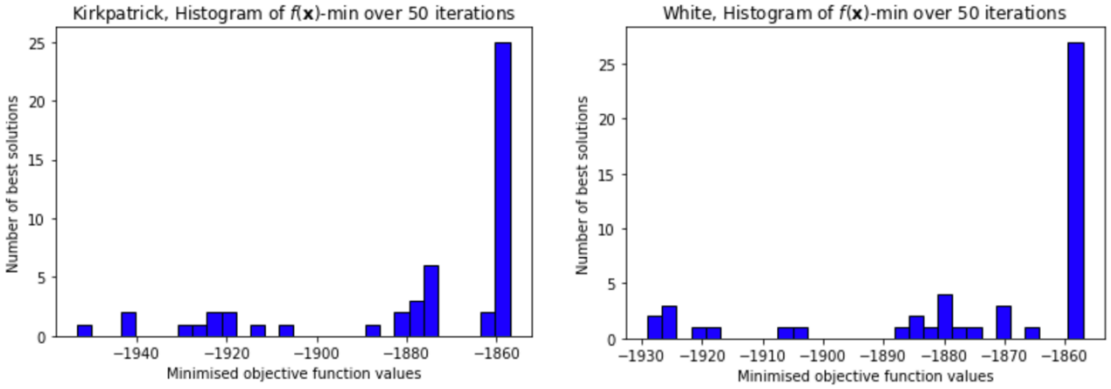


Figure 7. Histograms of best solution for two temperature schemes: Kirkpatrick (left) and White (right)

It can be deducted that the initial temperature method by White performs better.

#### 2.3.2 Temperature decrement (Huang vs ECS ($\alpha = 0.95$))

| | $f_{min}$ best | $f_{min}$ average | $\sigma_f, (s.d\ of\ f_{min})$ |
|---|---|---|---|
| Huang | -1929.00 | -1870.60 | 23.41 |
| ECS, **Control** | -1953.22 | -1875.59 | 27.21 |

Table 3. Effect of 2 different temperature decrement schemes on 5D-RF

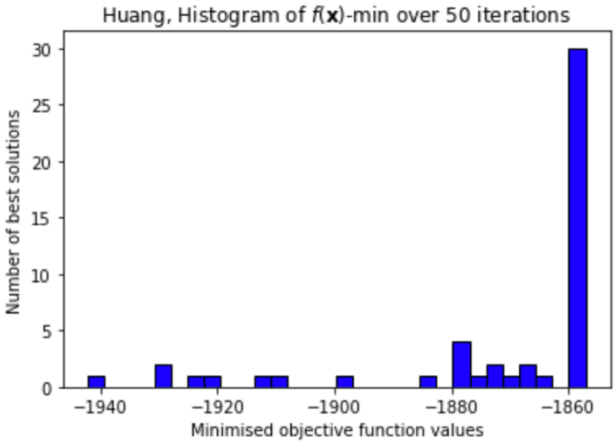


Figure 8. Histogram of best solution for Huang temperature decrement scheme

From Table 3, and comparing figure 8 to figure 7, it can be seen that using the Huang method preforms slightly worse than using the simple Exponential Cooling Scheme for temperature decrement. Using this method is also more computationally efficient.

### 2.3.3 $(k, L_k)$ Combinations

We vary $(k, L_k)$, while keeping $k \times L_k$ constant and below 10,000. Because 100 objective evaluations were already used in initial temperature determination, $k \times L_k$ is kept constant at a value of 9000.

| $(k, L_k)$ | $f_{min}$ best | $f_{min}$ average | $\sigma_f, (s.d\ of\ f_{min})$ |
|---|---|---|---|
| (10,900) | -2009.47 | -1871.92 | 34.27 |
| (20,450) | -1963.82 | -1869.86 | 21.78 |
| (30,300) **Control** | -1953.22 | -1875.59 | 27.21 |
| (40,225) | -1926.94 | -1870.64 | 21.83 |
| (50,180) | -1937.90 | -1868.73 | 20.91 |
| (60,150) | -1929.02 | -1866.39 | 19.25 |
| (70,129) | -1994.39 | -1881.51 | 32.37 |
| (80,113) | -1942.06 | -1874.96 | 25.76 |
| (90,100) | -1939.08 | -1874.29 | 25.44 |

Table 4. Effect of $(k, L_k)$ combinations on 5D-RF

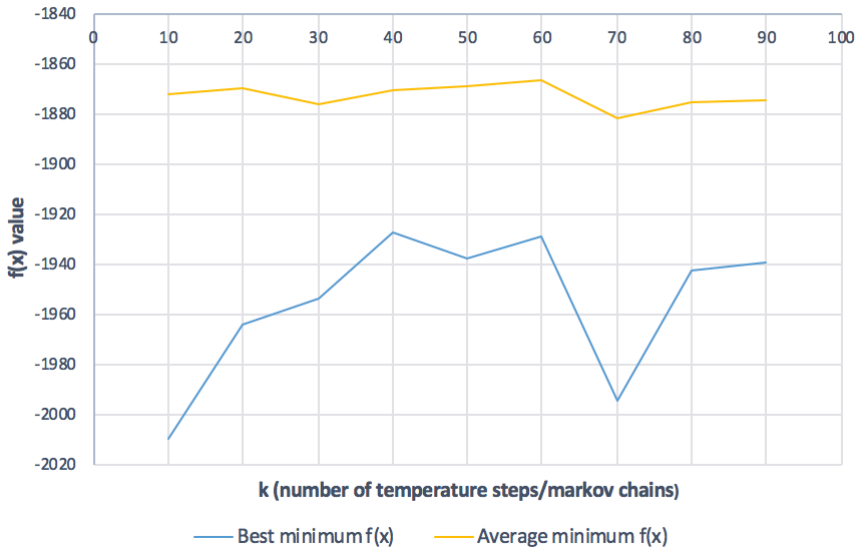


Figure 9. Plots of $k$ (number of Markov Chains) against best result obtained (blue), and the average result obtained (yellow)

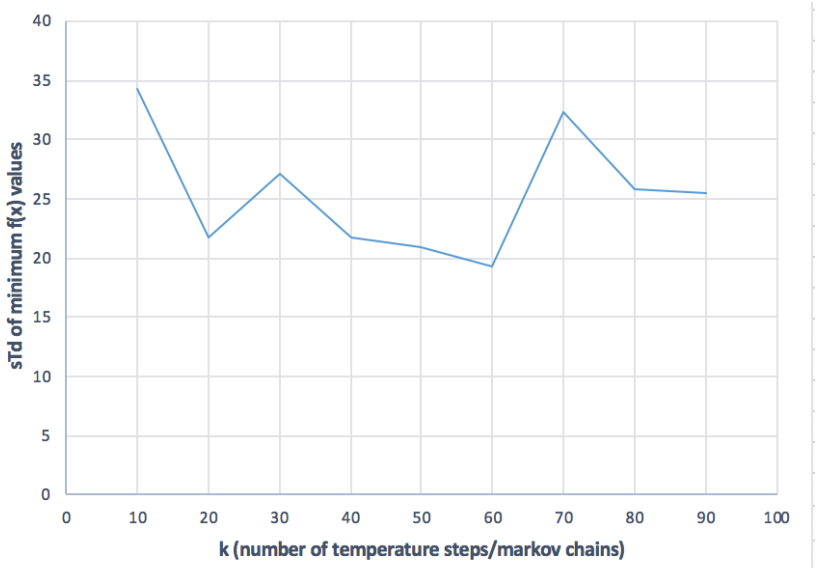


Figure 10. Plot of $k$ (number of Markov Chains) against the standard deviation of results over 50 iterations

It can be seen that the minimum average objective is obtained when $(k, L_k)$=(70,129). However, this combination of parameters comes at a price of having a high variance in results obtained.

# 3. Genetic Algorithm

As the second algorithm, the Genetic Algorithm was implemented to the 2D-RF for visualization of algorithm, and 5D-RF to compare the effects of parameter changes to algorithm performance. The GA attempts to simulate the phenomenon of natural evolution first observed by Darwin and Dawkins. The GA algorithm works by using a encoding of the control variables, $\boldsymbol{x}$, and search from one population of solutions to other. The operation of generating a new solution is conducted via selection, recombination and mutation. Continuous control variables are approximated by equivalent integer variables. [2]

## 3.1 Algorithm Description and implementation details

The one of the Algorithms proposed from the 4M17 Cambridge Moodle website: the Pypi-distributed python library called 'geneticalgorithm' was used for implementation [3]. The source code was found on Github and altered slightly for implementation and visualization. Although the algorithm has the option to implement the elitist scheme mentioned in lecture notes [2], we use the standard proportional selection method instead. The initial population was generated randomly.

As the population size, $N$, was varied, the number of iterations (generations) performed was adjusted accordingly to keep $N * n_{gen} = constant < 10{,}000$. The parameters to vary in the report were:

- Combinations of (Crossover probability, $P_c$, Mutation probability, $P_m$, population size $N$)
- Breeding Crossover method: one-point or two-point or uniform

## 3.2 2D-RF Genetic Algorithm

The parameters in table 5 show the control parameters used, unless otherwise stated and varied by different parts of the 5D-RF investigation. Again, the algorithm was run 50 times in each case, to increase reliability and observe the variation of results.

| Parameter | Value |
|---|---|
| $(N, P_c, P_m)$ | (400, 0.7, 0.006) |
| Crossover Method | One-point |

Table 5: Standard Parameters used for GA (Control)

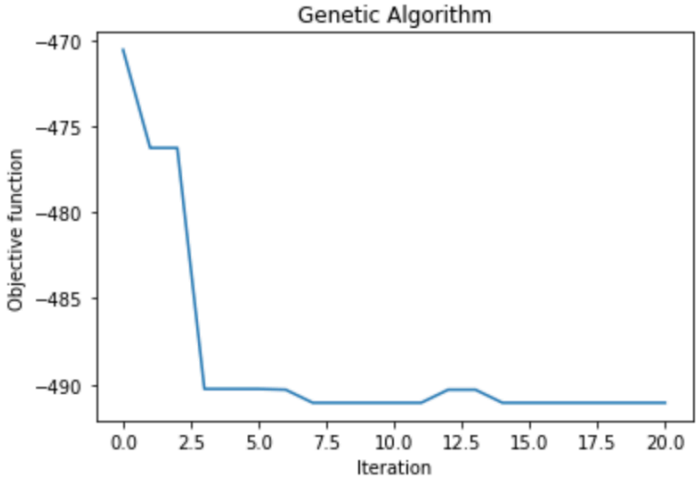


Figure 11. 2D-RF objective value evolution over generations, $(N, P_c, P_m) = (500, 0.7, 0.006)$

The code was altered to extract the x-coordinates at each generation, giving the path followed by Figure 11.

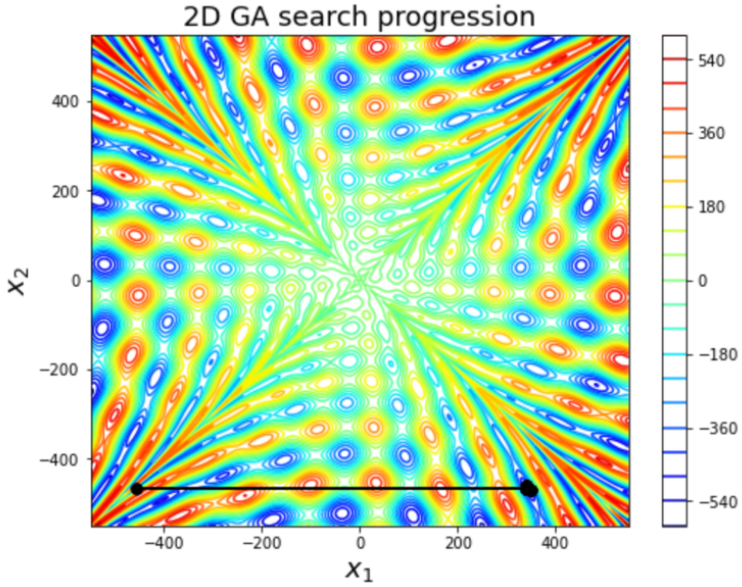


Figure 11. 2D-RF GA search progression on a contour map, $(N, P_c, P_m) = (500, 0.7, 0.006)$

From figure 11, it can be seen that GA will probably perform worse than SA. It can be seen that not a wide range is explored in its search, and due to the limited number of generation within the search (only 20 generations above), it showed a higher variance in results than in SA. Again, a histogram of results for 50 algorithm iterations was drawn to visualize the reliability of the algorithm.

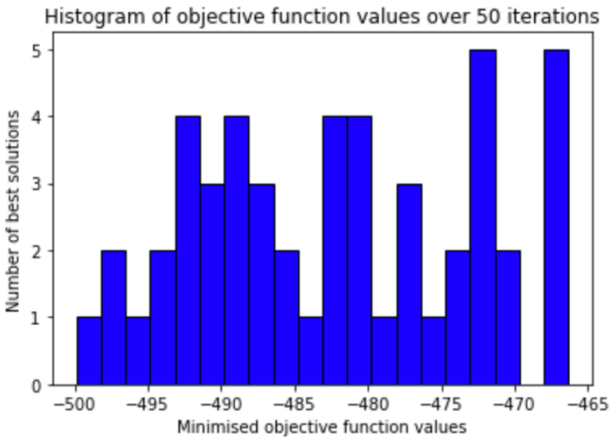


Figure 12. Histogram of minimum objective functions over 50 iterations of GA on 2D-RF

It can be seen that the GA shows a much higher variation of results than SA in the same number of iterations in a search. It is not clear from figure 10 where the minimum lies between -500 and -465, whereas in SA, it was much clearer that the global optimum was near -500.

### 3.3 5D-RF Genetic Algorithm

Using the GA on 5D-RF, the performance of different algorithm parameters was investigated. We vary 4 parameters: the population, $N$, the mutation probability, $P_m$, and the crossover probability, $P_c$, and the crossover type in that order, and we select the best parameter to use as our control as we proceed with the investigation.

#### 3.3.1 Varying '$N$', population size

| $(N, P_c, P_m)$ | $f_{min}$ best | $f_{min}$ average | $\sigma_f, (s.d\ of\ f_{min})$ |
|---|---|---|---|
| (5, 0.5,0.01) | -1833.84 | -1595.13 | 134.08 |
| (10, 0.5,0.01) | -1793.36 | -1593.18 | 128.76 |
| (20, 0.5,0.01) | -1830.40 | -1561.8 | 144.25 |
| (40, 0.5,0.01) | -1808.12 | -1516.75 | 151.22 |
| (50, 0.5,0.01) | -1821.39 | -1541.36 | 158.62 |
| (100, 0.5,0.01) | -1769.20 | -1498.22 | 139.42 |
| (200, 0.5,0.01) | -1835.03 | -1516.86 | 144.068 |
| (400, 0.5,0.01) | -1841.87 | -1608.03 | 101.23 |
| (500, 0.5,0.01) **Selected** | -1813.15 | -1581.27 | 114.56 |

Table 6. Effect of varying population size from 5 to 500

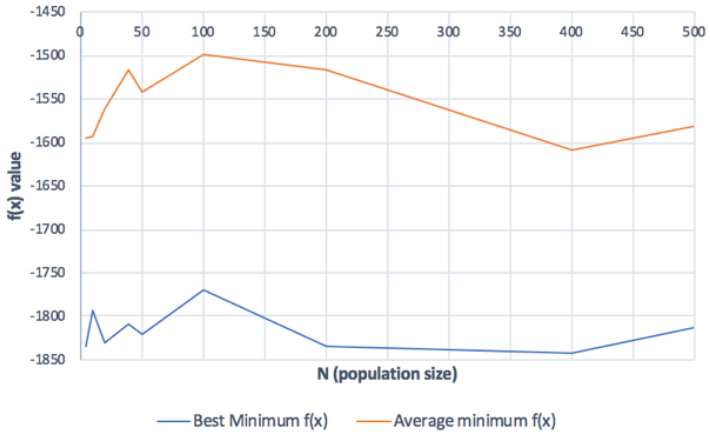


Figure 13. Plots of $N$ against best result obtained (blue), and the average result obtained (orange)

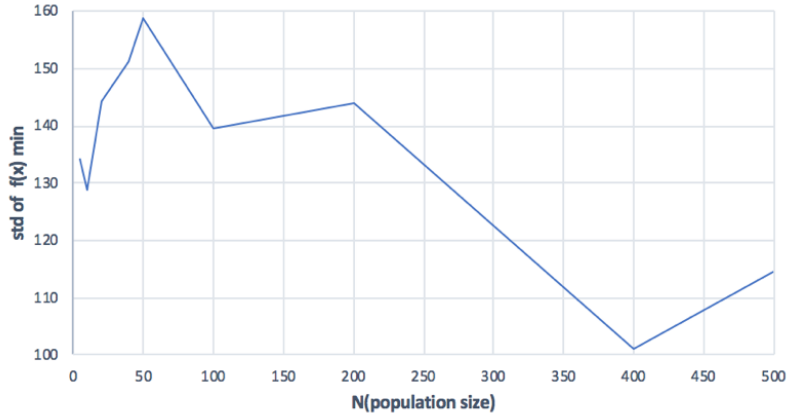


Figure 14. Plot of $N$ against standard deviation of minimum objective obtained over 50 iterations

It can be seen from figure 13 that the minimum solution is obtained for N=400. As it is believed that the figure shows a generally decreasing trend in results as N increased, N=500 was chosen as the optimal parameter. This is also efficient as a higher population size means less number of generations and higher computational efficiency.

### 3.3.2 Varying '$P_m$', Mutation Probability

| $(N, P_c, P_m)$ | $f_{min}$ best | $f_{min}$ average | $\sigma_f, (s.d\ of\ f_{min})$ |
|---|---|---|---|
| (500, 0.5,0.001) | -1728.10 | -1569.72 | 83.50 |
| (500, 0.5,0.002) | -1872.15 | -1609.38 | 104.83 |
| (500, 0.5,0.004) | -1827.70 | -1594.01 | 105.81 |
| (500, 0.5,0.006) **Selected** | -1807.90 | -1615.84 | 97.86 |
| (500, 0.5,0.008) | -1933.66 | -1629.87 | 105.16 |
| (500, 0.5,0.01) | -1813.15 | -1581.27 | 114.56 |
| (500, 0.5,0.02) | -1818.34 | -1615.46 | 101.21 |

Table 7. Effect of varying mutation probability from 0.001 to 0.02

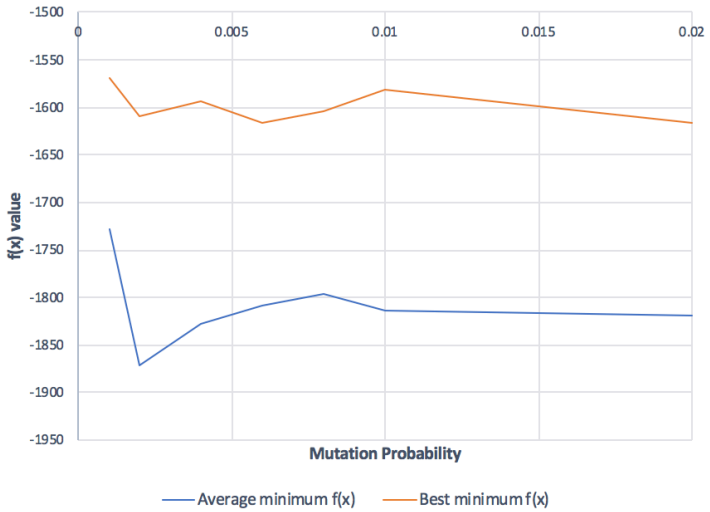


Figure 15. Plots of mutation probability against best result obtained (blue), and the average result obtained (orange)

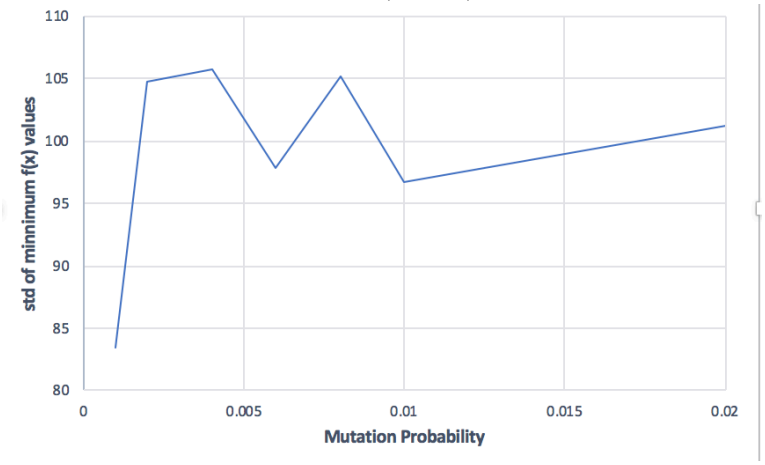


Figure 16. Plot of mutation probability against standard deviation of minimum objective obtained over 50 iterations

The minimum average objective is obtained when $P_m$= 0.006 as it can be seen inn figure 15. At this point, the standard deviation of results is also not too significant. As a decreasing trend is observed as mutation probability succeeds 0.1, perhaps allowing too may populations to mutate is not good in preserving significant features in the population.

### 3.3.3 Varying '$P_c$', Crossover Probability

| $(N, P_c, P_m)$ | $f_{min}$ best | $f_{min}$ average | $\sigma_f, (s.d\ of\ f_{min})$ |
|---|---|---|---|
| (500, 0.1,0.006) | -1832.51 | -1458.05 | 142.92 |
| (500, 0.2,0.006) | -1761.71 | -1507.80 | 117.41 |
| (500, 0.3,0.006) | -1855.18 | -1582.63 | 118.95 |
| (500, 0.4,0.006) | -1761.66 | -1593.50 | 100.93 |
| (500, 0.5,0.006) | -1745.09 | -1533.43 | 109.76 |
| (500, 0.6,0.006) | -1813.05 | -1620.44 | 93.64 |
| (500, 0.7,0.006) **Selected** | -1840.58 | -1621.64 | 88.27 |
| (500, 0.8,0.006) | -1839.27 | -1621.35 | 101.96 |
| (500, 0.9,0.006) | -1834.41 | -1633.17 | 114.05 |
| (500, 1 ,0.006) | -1795.71 | -1640.06 | 92.21 |

Table 8. Effect of varying crossover probability between 0.1 and 1

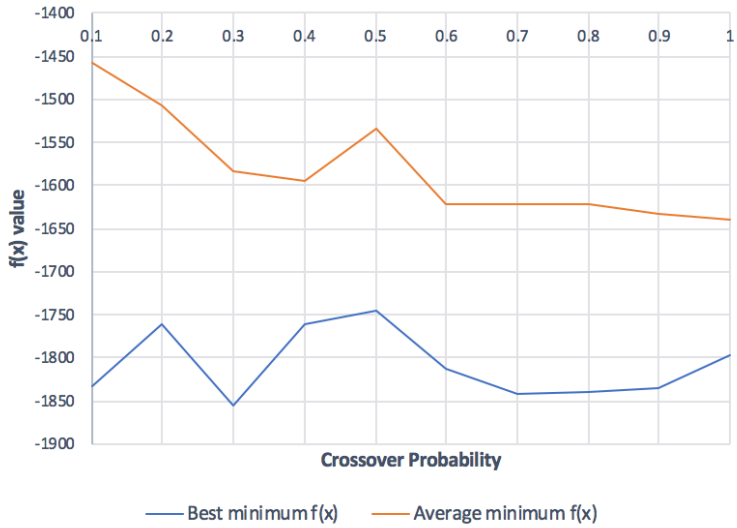


Figure 17. Plots of Crossover probability against best result obtained (blue), and the average result obtained (orange)

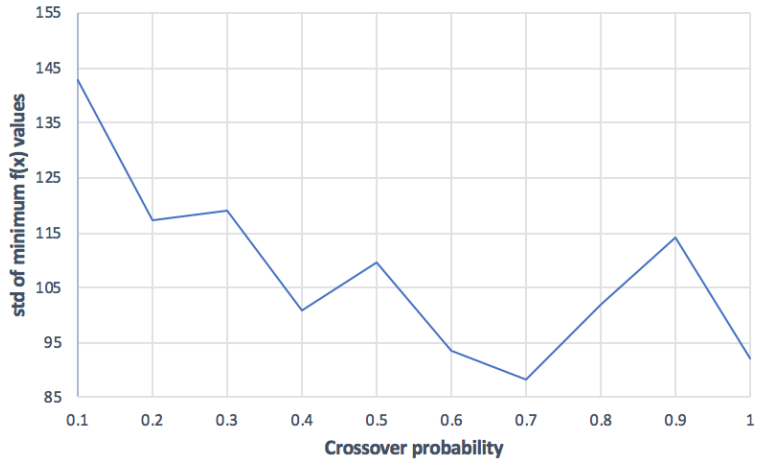


Figure 18. Plot of Crossover probability against standard deviation of minimum objective obtained over 50 iterations

It can be seen from figure 17 that the average solution shows a generally decreasing trend up to probabilities of 0.6 as crossover probability increases. As figure 18 shows that the variance of results is significantly low when $P_c$=0.7, this probability was chosen as our best attempt. Ultimately, from the first 3 parameter investigations, $(N, P_c, P_m)$= (500, 0.7,0.006) was chosen as our best trial.

### 3.3.4 Crossover Type: one-point or two-point

Describe from lecture notes: and describe what uniform is.

| Crossover Type | $f_{min}$ best | $f_{min}$ average | $\sigma_f, (s.d\ of\ f_{min})$ |
|---|---|---|---|
| One-point | -1840.58 | -1621.64 | 88.27 |
| Two-points | -1906.54 | -1596.61 | 115.11 |
| Uniform | -1769.44 | -1605.27 | 94.26 |

Table 9. Effect of crossover type in GA

Comparing the average of the solutions, it can be deducted that the one-point crossover type is best for 5D-RF.

## 4. Algorithm Comparison

Having chosen our best attempt as the parameters for SA ad GA, we now compare the results of 2D-RF and 5D-RF optimizations using these parameters. One anomaly in the comparison is that in SA, we have used 9130 objective function evaluations, whereas in GA, we have used 10,000.
The parameters used for SA are: White's method for Temperature initialization, ECS for temperature decrement, and $(k, L_k)$=(70,129).
The parameters used for GA are: $(N, P_c, P_m)$= (500, 0.7,0.006), and one-point crossover method.

| Algorithm | $f_{min}$ best | $f_{min}$ average | $\sigma_f, (s.d\ of\ f_{min})$ |
|---|---|---|---|
| SA-2DRF | -500.80 | -499.28 | 1.88 |
| GA- 2DRF | -500.21 | -481.42 | 11.07 |
| SA-5DRF | -1928.30 | -1872.75 | 21.2 |
| GA- 5DRF | -1840.58 | -1621.64 | 88.27 |

Table 10. Results of SA and GA on 2D and 5D-RF using best parameters selected

Even though the number of objective evaluations was smaller for SA implementation, SA significantly outperforms GA for both 2D-RF and 5D-RF objective minimization. This is as expected, because it is well known that Genetic algorithms are designed to be used with discrete control variables, and do not perform well with continuous ones as we have used here. The algorithmic performance of Evolution Strategies; an algorithm that also simulates natural evolution; would have been better, as it is designed to be applied to continuous parameter optimization problems. But GA computationally more efficient than SA. Unlike SA, which is a sequential program, GAs are well-suited to implementation on parallel computers, where objective function and constraint evaluations can be done simultaneously. SA performs way better than GA.

## 5. Conclusion

It can be concluded from the investigation that in the global objective minimization of highly complicated continuous-variable functions, like the Rana Function, the Simulated Annealing Algorithm outperforms the Genetic Algorithm without dispute. However, if the constraint was to keep the computational time or cost equivalent, rather than the number of objective evaluations to compare the two algorithms, results may have been different. This is because GA computationally more efficient than SA. Unlike SA, which is a sequential program, GAs are well-suited to implementation on parallel computers, where objective function and constraint evaluations can be done simultaneously. The algorithmic performance would have been better if we had used Evolution Strategies instead of GA, which is an algorithm that also simulates natural evolution and is computationally efficient also, but it is designed to be applied to continuous parameter optimization problems.

# 6. Appendix A: Code

## A.1 Simulated Annealing 2D-RF

```
import numpy as np
import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
np.random.seed()
```

```
#define Rana 2D objective function
def rana2D(x1,x2):
    y= (x1* np.cos(np.sqrt(abs(x2+x1+1)))*np.sin(np.sqrt(abs(x2-x1+1))))
    \+ ((1+x2)*np.cos(np.sqrt(abs(x2-x1+1)))*np.sin(np.sqrt(abs(x2+x1+1))))
    return y

#upscale x for update step, such that it varies between -500&500 instead o
def upscale_x(x):
    x_upscaled = 500*(2*x -1)
    return x_upscaled

#define a function that clips x values outside 0 to 1
def clip(x):
    if x>1:
        return 1
    if x<0:
        return 0
    else:
        return x
```

```
# generate new trial solutions by Parks, 1990
def gen_x_parks(x,D,T):
    #work with x between 0 and 1 to start with
    D_new= np.zeros((2,2))
    x_new= np.zeros(2)
    R_kk=np.array([D[0][0]*(np.random.uniform(-1,1)) , \
                   D[1][1]*(np.random.uniform(-1,1))])
    #R=changes made to each x, so add Rkk to each x
    x_new[0]=clip(x[0]+R_kk[0])
    x_new[1]=clip(x[1]+R_kk[1])

    #change in obj fn value
    df= rana2D(upscale_x(x_new[0]),upscale_x(x_new[1])) - \
    rana2D(upscale_x(x[0]),upscale_x(x[1]))
    #actual step size:
    d= np.sqrt(R_kk[0]**2+ R_kk[1]**2)
    R_kk= np.fabs(R_kk)
    #update D
    D_new= np.zeros((2,2))
    D_new[0][0]= (1-damp)*D[0][0] + damp*weight*abs(R_kk[0])
    D_new[1][1]= (1-damp)*D[1][1] + damp*weight*abs(R_kk[1])

    #probability of accepting an increase in f
    p= np.exp(-df/(T*d))
    #decrease in f, just accept
    if df<0:
        return x_new, D_new
    #accept with prob p if f increases
    elif p > np.random.uniform(0,1):
        return x_new, D_new

    else:
        return x, D
```

```
def T0_estimate_white(x,D,Lk_init):
    #by white 1984, T0= std of list of f values observed in initial search
    #Lk_init= number of initial search trials

    f_list=[]

    for i in range(0,Lk_init):
        D_new= np.zeros((2,2))
        x_new= np.zeros(2)
        R_kk=np.array([D[0][0]*np.random.uniform(-1,1) ,\
                       D[1][1]*np.random.uniform(-1,1)])

        x_new[0]=clip(x[0]+R_kk[0])
        x_new[1]=clip(x[1]+R_kk[1])
        R_kk= np.fabs(R_kk)

        D_new[0][0]= (1-damp)*D[0][0] + damp*weight*R_kk[0]
        D_new[1][1]= (1-damp)*D[1][1] + damp*weight*R_kk[1]

        #list of new f values, needed for T0 estimation
        f= rana2D(upscale_x(x_new[0]),upscale_x(x_new[1]))
        f_list.append(f)
        #update x and D also?
        x= x_new
        D= D_new

    T0= np.std(f_list)
    return T0
```

```
# SA combined:
Lk_init=100


def SA_by_MC(k,Lk):
#k= number of Steps(MCs), Lkk=length of each step (MCs)
    f_list=[]
    x_list=[]
    #x0 generated randomly from seed
    x0= np.array([np.random.rand(),np.random.rand()])
    D0= np.zeros((2,2))
    D0[0][0]= x0[0]
    D0[1][1]= x0[1]
    #initialise T
    T= T0_estimate_white(x0,D0,Lk_init)
    #first update of x and D
    x,D= gen_x_parks(x0,D0,T)
    #iterate
    for i in range(0,k):
        for j in range(0,Lk):
            XX,DD= gen_x_parks(x,D,T)
            if np.any(x != XX)== True:
                x,D = XX,DD
                f_list.append(rana2D(upscale_x(x[0]),upscale_x(x[1])))
                x_list.append((upscale_x(x[0]),upscale_x(x[1])))
        #T decrement outside MC
        T*=0.95

    f_min=min(f_list)
    x_coords=x_list[f_list.index(f_min)]

    return f_list, x_list, f_min, x_coords
```

```
#define params
damp=0.1
weight=2.1
Lk_init=100
k= 30
Lk= 300
num_iter=50 #run SA 50 times

#visualise function:
xvalues = yvalues = np.arange(-550, 550, 1)
X, Y = np.meshgrid(xvalues, yvalues)

zs = np.array([rana2D(x,y) for x,y in zip(np.ravel(X), np.ravel(Y))])
Z = zs.reshape(X.shape)
```

```
#figures:

fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, Z, rstride=5, cstride=5, linewidth=0,\
                       cmap=cm.jet)
ax.set_title('Surface Plot of 2D Rana Function', fontsize=18)
ax.set_xlabel('$x_1$', fontsize=14)
ax.set_ylabel('$x_2$', fontsize=14)
ax.set_zlabel('$f(\mathbf{x})$', fontsize=14)
plt.tight_layout()

plt.show()

fig2 = plt.figure(figsize=(8,6))
ax = fig2.add_subplot(1,1,1)
plt.contour(X, Y, Z, 20, cmap=cm.jet)
ax.set_title('Contour Plot of 2D Rana Function', fontsize=14)
ax.set_xlabel('$x_1$', fontsize=14)
ax.set_ylabel('$x_2$', fontsize=14)
cbar = fig2.colorbar(surf, aspect=14)
cbar.set_label('$f(\mathbf{x})$', rotation=0, fontsize=14)
plt.show()
```

```
  one_result=SA_by_MC(k,Lk)
  print(one_result[2],one_result[3])
```

```
# plots for one search:
coords= one_result[1]
best_coords= one_result[3]
x1_coords= []
x2_coords= []
for i in range (0,len(coords)):
    x1_coords.append(coords[i][0])
    x2_coords.append(coords[i][1])
```

```
fig3 = plt.figure(figsize=(8,6))
ax = fig3.add_subplot(1,1,1)
plt.contour(X, Y, Z, 20, cmap=cm.jet)
ax.scatter(x1_coords,x2_coords, color='black',s=30, zorder=1000)
#plt.plot(x1_coords,x2_coords,color='black')
ax.set_title('2D SA search progression', fontsize=18)
ax.set_xlabel('$x_1$', fontsize=18)
ax.set_ylabel('$x_2$', fontsize=18)
plt.colorbar()
cbar.set_label('$f(\mathbf{x})$', rotation=0, fontsize=18)
plt.show()

fig4 = plt.figure(figsize=(8,6))
ax = fig4.add_subplot(1,1,1)
ax.scatter(best_coords[0],best_coords[1],color='black', s=100,zorder=1000)
plt.contour(X, Y, Z, 20, cmap=cm.jet,zorder=2)
ax.set_title('Visualise best solution position', fontsize=18)
ax.set_xlabel('$x_1$', fontsize=18)
ax.set_ylabel('$x_2$', fontsize=18)
plt.colorbar()
cbar.set_label('$f(\mathbf{x})$', rotation=0, fontsize=18)
plt.show()
```

```
fig5 = plt.figure(figsize=(8,6))
ax = fig5.add_subplot(1, 1, 1)
a_list=list(range(len(one_result[0])))
a2_list= a_list[::10]
b_list= one_result[0]
b2_list= b_list[::10]
ax.scatter(a2_list, b2_list,s=10)
plt.xlabel('evaluation', Fontsize=14)
plt.ylabel('$f(\mathbf{x})$', Fontsize=14)
plt.title('Objective value evolution of 2D RF')
```

```
obj_fn_list=[]
f_min_list=[]
x_coords_list=[]
for i in range(0,num_iter):
    RESULTS=SA_by_MC(k,Lk)
    f_min_list.append(RESULTS[2])
    x_coords_list.append(RESULTS[3])
    obj_fn_list.append(RESULTS[0])
f_min_final= min(f_min_list)
x_coord_final= x_coords_list[f_min_list.index(f_min_final)]

print(f_min_final, x_coord_final)
```

```
obj_fn_list=[]
f_min_list=[]
x_coords_list=[]
for i in range(0,num_iter):
    RESULTS=SA_by_MC(k,Lk)
    f_min_list.append(RESULTS[2])
    x_coords_list.append(RESULTS[3])
    obj_fn_list.append(RESULTS[0])
f_min_final= min(f_min_list)
x_coord_final= x_coords_list[f_min_list.index(f_min_final)]

print(f_min_final, x_coord_final)
```

```
#plot histogram of f_min_list over num_iter iterations
plt.hist(f_min_list, 30, color='blue', histtype='bar', ec='black')
plt.title('Histogram of objective function values over 50 iterations')
plt.xlabel('Minimised objective function values')
plt.ylabel('Number of best solutions')
plt.show()
```

```
print(np.std(f_min_list))
print(np.mean(f_min_list))
print(min(f_min_list))
```

### A.2 Simulated Annealing 5D-RF

```
def rana5D(x1,x2,x3,x4,x5):
    y1= (x1* np.cos(np.sqrt(abs(x2+x1+1))) *np.sin(np.sqrt(abs(x2-x1+1))))
    y2= (x2* np.cos(np.sqrt(abs(x3+x2+1))) *np.sin(np.sqrt(abs(x3-x2+1))))
    y3= (x3* np.cos(np.sqrt(abs(x4+x3+1))) *np.sin(np.sqrt(abs(x4-x3+1))))
    y4= (x4* np.cos(np.sqrt(abs(x5+x4+1))) *np.sin(np.sqrt(abs(x5-x4+1))))
    return y1+y2+y3+y4

def upscale_x(x):
    x_upscaled = 500*(2*x -1)
    return x_upscaled

def clip(x):
    if x>1:
        return 1
    if x<0:
        return 0
    else:
        return x
```

```
# generate new trial solutions by Parks, 1990
def gen_x_parks(x,D,T):

    D_new= np.zeros((5,5))
    x_new= np.zeros(5)
    R_kk=np.array([D[0][0]*(np.random.uniform(-1,1)) , \
                   D[1][1]*(np.random.uniform(-1,1)),\
                   D[2][2]*(np.random.uniform(-1,1)),\
                   D[3][3]*(np.random.uniform(-1,1)),\
                   D[4][4]*(np.random.uniform(-1,1))])
    x_new[0]=clip(x[0]+R_kk[0])
    x_new[1]=clip(x[1]+R_kk[1])
    x_new[2]=clip(x[2]+R_kk[2])
    x_new[3]=clip(x[3]+R_kk[3])
    x_new[4]=clip(x[4]+R_kk[4])

    #change in obj fn value
    df= rana5D(upscale_x(x_new[0]),upscale_x(x_new[1]),upscale_x(x_new[2]),
    #actual step size:
    d= np.sqrt(R_kk[0]**2+ R_kk[1]**2+R_kk[2]**2+R_kk[3]**2+R_kk[4]**2)
    R_kk= np.fabs(R_kk)
    #update D
    D_new= np.zeros((5,5))
    D_new[0][0]= (1-damp)*D[0][0] + damp*weight*abs(R_kk[0])
    D_new[1][1]= (1-damp)*D[1][1] + damp*weight*abs(R_kk[1])
    D_new[2][2]= (1-damp)*D[2][2] + damp*weight*abs(R_kk[2])
    D_new[3][3]= (1-damp)*D[3][3] + damp*weight*abs(R_kk[3])
    D_new[4][4]= (1-damp)*D[4][4] + damp*weight*abs(R_kk[4])
    #probability of accepting an increase in f
    p= np.exp(-df/(T*d))
    #decrease in f, just accept
    if df<0:
        return x_new, D_new
    #accept with prob p if f increases
    elif p > np.random.uniform(0,1):
        return x_new, D_new
    else:
        return x, D
```

```
def T0_estimate_white(x,D,Lk_init):
    #by white 1984, T0= std of list of f values observed in initial search
    #Lk_init= number of initial search trials

    f_list=[]

    for i in range(0,Lk_init):
        D_new= np.zeros((5,5))
        x_new= np.zeros(5)
        R_kk=np.array([D[0][0]*(np.random.uniform(-1,1)) ,\
                       D[1][1]*(np.random.uniform(-1,1)),\
                       D[2][2]*(np.random.uniform(-1,1)),\
                       D[3][3]*(np.random.uniform(-1,1)),\
                       D[4][4]*(np.random.uniform(-1,1))])

        x_new[0]=clip(x[0]+R_kk[0])
        x_new[1]=clip(x[0]+R_kk[1])
        x_new[2]=clip(x[2]+R_kk[2])
        x_new[3]=clip(x[3]+R_kk[3])
        x_new[4]=clip(x[4]+R_kk[4])

        R_kk= np.fabs(R_kk)

        D_new[0][0]= (1-damp)*D[0][0] + damp*weight*R_kk[0]
        D_new[1][1]= (1-damp)*D[1][1] + damp*weight*R_kk[1]
        D_new[2][2]= (1-damp)*D[2][2] + damp*weight*abs(R_kk[2])
        D_new[3][3]= (1-damp)*D[3][3] + damp*weight*abs(R_kk[3])
        D_new[4][4]= (1-damp)*D[4][4] + damp*weight*abs(R_kk[4])
        #list new f values, needed for T0 estimation
        f= rana5D(upscale_x(x_new[0]),upscale_x(x_new[1]),\
                  upscale_x(x_new[2]),upscale_x(x_new[3]),\
                  upscale_x(x_new[4]))
        f_list.append(f)
        x= x_new
        D= D_new

    T0= np.std(f_list)
    return T0
```

```
# SA combined:
Lk_init=100
def SA_by_parks(k,Lk):
#k= number of Steps(MCs), Lkk=length of each step (MCs)
    f_list=[]
    x_list=[]
    #x0 generated randomly from seed
    x0= np.array([np.random.rand(),np.random.rand(),np.random.rand(),\
                  np.random.rand(),np.random.rand()])
    D0= np.zeros((5,5))
    D0[0][0]= x0[0]
    D0[1][1]= x0[1]
    D0[2][2]= x0[2]
    D0[3][3]= x0[3]
    D0[4][4]= x0[4]
    #initialise T
    T= T0_estimate_white(x0,D0,Lk_init)
    #first update of x and D
    x,D= gen_x_parks(x0,D0,T)
    #iterate
    for i in range(0,k):
        for j in range(0,Lk):
            XX,DD= gen_x_parks(x,D,T)
            if np.any(x != XX)== True:
                x,D = XX,DD
                f_list.append(rana5D(upscale_x(x[0]),\
                                     upscale_x(x[1]),\
                                     upscale_x(x[2]),\
                                     upscale_x(x[3]),\
                                     upscale_x(x[4])))
                x_list.append((upscale_x(x[0]),\
                               upscale_x(x[1]),\
                               upscale_x(x[2]),\
                               upscale_x(x[3]),\
                               upscale_x(x[4])))
        #T decrement outside MC
        T*= 0.95
```

```
    f_min=min(f_list)
    x_coords=x_list[f_list.index(f_min)]

    return f_list, x_list, f_min, x_coords
```

```
#define params
damp=0.1
weight=2.1
Lk_init=100
k= 70
Lk= 129
num_iter=50 #run SA 50 times
```

```
obj_fn_list=[]
f_min_list=[]
x_coords_list=[]
for i in range(0,num_iter):
    RESULTS=SA_by_parks(k,Lk)
    f_min_list.append(RESULTS[2])
    x_coords_list.append(RESULTS[3])
    obj_fn_list.append(RESULTS[0])
f_min_final= min(f_min_list)
x_coord_final= x_coords_list[f_min_list.index(f_min_final)]

print(f_min_final, x_coord_final)
```

```
print(np.std(f_min_list))
print(np.mean(f_min_list))
```

### A.3 GA: geneeticalgorithm.py

Reference [3] python file edited to output desired population during search for 2D path plot

### A.4 2D-RF GA implementation on Jupyter Notebook

```
import numpy as np
import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from geneticalgorithm import geneticalgorithm as ga
```

```
#2D: 1 iter
def rana2D(x):
    y=0
    for i in range (0,1):
        y += (x[i]* np.cos(np.sqrt(abs(x[i+1]+x[i]+1))) \
              *np.sin(np.sqrt(abs(x[i+1]-x[i]+1)))) + ((1+x[i+1]) \
                * np.cos(np.sqrt(abs(x[i+1]-x[i]+1))) \
                           *np.sin(np.sqrt(abs(x[i+1]+x[i]+1))))
    return y

varbound=np.array([[-500,500]]*2)
algorithm_param= {'max_num_iteration': 20, \
                  'population_size': 500, \
                  'mutation_probability': 0.006, \
                  'elit_ratio': 0, \
                  'crossover_probability': 0.7, \
                  'parents_portion': 0.3, \
                  'crossover_type': 'one_point', \
                  'max_iteration_without_improv': None}
model=ga(function=rana2D,dimension=2, \
         variable_type='real',variable_boundaries=varbound, \
         algorithm_parameters=algorithm_param)
model.run()
```

```
fmin_values=model.report#list including the convergence of algorithm over i
best_coords=model.variables
solution=model.output_dict #dict includign best set of variables found and
print(best_coords)
print(solution)
print(len(best_coords))
x1_coords= []
x2_coords= []
for i in range (0,len(best_coords)):
    x1_coords.append(best_coords[i][0])
    x2_coords.append(best_coords[i][1])
```

```
xvalues = yvalues = np.arange(-550, 550, 1)
X, Y = np.meshgrid(xvalues, yvalues)

zs = np.array([rana2D([x,y]) for x,y in zip(np.ravel(X), np.ravel(Y))])
Z = zs.reshape(X.shape)
```

```
#2D: 50 iter
def rana2D(x):
    y=0
    for i in range (0,1):
        y += (x[i]* np.cos(np.sqrt(abs(x[i+1]+x[i]+1))) \
              *np.sin(np.sqrt(abs(x[i+1]-x[i]+1)))) + \
        ((1+x[i+1]) * np.cos(np.sqrt(abs(x[i+1]-x[i]+1))) \
         *np.sin(np.sqrt(abs(x[i+1]+x[i]+1))))
    return y

varbound=np.array([[-500,500]]*2)
algorithm_param= {'max_num_iteration': 50, \
                  'population_size': 200, \
                  'mutation_probability': 0.006, \
                  'elit_ratio': 0, \
                  'crossover_probability': 0.7, \
                  'parents_portion': 0.3, \
                  'crossover_type': 'one_point', \
                  'max_iteration_without_improv': None}
fmin_list=[]
fmin_best=[]
for i in range(0, 50):
    model2=ga(function=rana2D,dimension=2, variable_type='real',\
              variable_boundaries=varbound, \
              algorithm_parameters=algorithm_param)
    model2.run()
    fmin_list.append(model2.report)
    fmin_best.append(model2.best_function)
```

```
print(np.std(fmin_best))
print(np.mean(fmin_best))
print(min(fmin_best))
```

## A.4 5D-RF GA implementation Jupyter Notebook

```
import numpy as np
import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from geneticalgorithm import geneticalgorithm as ga
```

```
#5D:
def rana5D(x):
    y=0
    for i in range (0,4):
        y += (x[i]* np.cos(np.sqrt(abs(x[i+1]+x[i]+1))) \
              *np.sin(np.sqrt(abs(x[i+1]-x[i]+1)))) + \
        ((1+x[i+1]) * np.cos(np.sqrt(abs(x[i+1]-x[i]+1))) \
         *np.sin(np.sqrt(abs(x[i+1]+x[i]+1))))
    return y
#1 iteration
varbound=np.array([[-500,500]]*5)
algorithm_param= {'max_num_iteration': 500, \
                  'population_size': 20, \
                  'mutation_probability': 0.01, \
                  'elit_ratio': 0, \
                  'crossover_probability': 0.5, \
                  'parents_portion': 0.3, \
                  'crossover_type': 'one_point', \
                  'max_iteration_without_improv': None}
model2=ga(function=rana5D,dimension=5, \
          variable_type='real',variable_boundaries=varbound,\
          algorithm_parameters=algorithm_param)
model2.run()
```

```
fmin_values=model2.report#list including the convergence of algorithm over
solution=model2.output_dict #dict includign best set of variables found an
```

```
print(np.std(fmin_values))
print(np.mean(fmin_values))
```

```
#50 iterations
num_iter=50
varbound=np.array([[-500,500]]*5)
algorithm_param= {'max_num_iteration': 20, \
                  'population_size': 500, \
                  'mutation_probability': 0.006, \
                  'elit_ratio': 0, 'crossover_probability': 0.7, \
                  'parents_portion': 0.3, 'crossover_type': 'one_point', \
                  'max_iteration_without_improv': None}
```

```
fmin_list=[]
fmin_best=[]
for i in range(0, num_iter):
    model2=ga(function=rana5D,dimension=5, \
              variable_type='real',variable_boundaries=varbound, \
              algorithm_parameters=algorithm_param)
    model2.run()
    fmin_list.append(model2.report)
    fmin_best.append(model2.best_function)
```

```
print(np.std(fmin_best))
print(np.mean(fmin_best))
print(min(fmin_best))
```

## 7. References


[1] 4M17 Simulated Annealing Lecture Notes 2020-2021, GTP
https://www.vle.cam.ac.uk/pluginfile.php/11351401/mod_resource/content/1/4M17SimulatedAnnealing.pdf

[2] 4M17 Genetic Algorithm Lecture Notes 2020-2021, GTP
https://www.vle.cam.ac.uk/pluginfile.php/1735021/mod_resource/content/2/4M17GeneticAlgorithms.pdf

[3] Pypi Genetic Algorithm library, rmsolgi, Github
https://github.com/rmsolgi/geneticalgorithm