CUED - Engineering Tripos Part IIB 2017-2018 **Module Coursework**

Module	4M17	Title of report	**Assignment 2: Optimisation Algorithm Performance Comparison Study**

Date submitted: Jan 16th 2018	Assessment for this module is ☑ 100% / ☐ 25% coursework of which this assignment forms **50%**

UNDERGRADUATE STUDENTS ONLY		**POST GRADUATE STUDENTS ONLY**			
Candidate number:		Name:	E526Z	College:	Darwin

Feedback to the student

☐ **See also comments in the text**

		Very good	**Good**	Needs imprvmt
CONTENT	**Completeness, quantity of content:** Has the report covered all aspects of the assignment? Has the analysis been carried out thoroughly?			
	Correctness, quality of content Is the data correct? Is the analysis of the data correct? Are the conclusions correct?			
	Depth of understanding, quality of discussion Does the report show a good technical understanding? Have all the relevant conclusions been drawn?			
	Comments:			
PRESENTATION	**Attention to detail, typesetting and typographical errors** Is the report free of typographical errors? Are the figures/tables/references presented professionally?			
	Comments:			

Overall assessment (circle grade)		A*	**A**	B	C	D
Guideline standard		>75%	**65-75%**	55-65%	40-55%	<40%
Penalty for lateness:		*20% of marks per week or part week that the work is late.*				

Marker: Date:

Contents

Chapter 1

Introduction

The aim of this assignment was use two different methods to optimise the Eggholder function, defined in D dimensions as follows:

$$f(\mathbf{x}) = \sum_{i=1}^{D-1} \left[-(x_{i+1} + 47) \sin\left(\sqrt{|x_{i+1} + \frac{1}{2}x_i + 47|}\right) - x_i \sin\left(\sqrt{|x_i - x_{i+1} - 47|}\right) \right] \tag{1.1}$$

subject to:

$$-512 \leq x_i \leq 512 \tag{1.2}$$

The two methods applied were Simulated Annealing (SA) and a Genetic Algorithm(GA). For convenience both algorithms will be referred to by their initials SA and GA for the remainder of this report. The Eggholder function will be referred to by either 2DEH or 5DEH (depending on the relevant number of dimensions). Both algorithms were coded from scratch.

Chapter 2

Simulated Annealing

The SA algorithm was built according to the basic structure given in the lecture notes [1]. New solutions were generated according to the strategy attributed to Parks[1990] on the lower half of page 3 of the notes. The initial temperature for a given application of SA was generated according to the scheme attributed to White[1984] on the top half of page 6 of the lecture notes, where T is given by the standard deviation of the variance in the objective function if all solutions are accepted (for 100 solutions). The length of the markov chains were set as constant. Two methods for temperature decrementation were attempted: one with a constant coefficient ($\alpha = 0.95$) and one attributed to Huang *et al.* [1986] around half way down page 7 of the lecture notes.

The main parameters of interest were the number of markov chains in the SA schedule (k), the number of trials at each new temperature value (L_k) and α_k, the coefficient by which the temperature is decremented at the end of each markov chain. The cases of $\alpha_k = 0.95$ and $\alpha_k = \max[0.5, \exp(\frac{-0.7T_k}{\sigma_k}]$ were investigated, where σ_k is the standard deviation of the objective function values accepted at temperature T_k.

2.1 2D Eggholder function

The algorithm was first tested on the Eggholder function in 2 dimensions(2DEH).

A heat map of the function in 2 dimensions is given below:

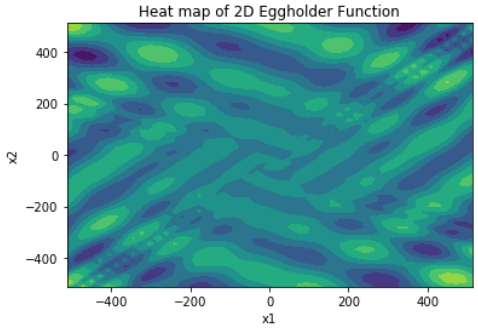

Figure 2.1: Heatmap of the 2DEH

A scatter plot of the points on the search space explored for a case where the SA arrived at close to the global minimum (-957.4963, x1, x2 = 512.0, 405.60, parameters:($k = 20$, $L_k = 500$, $\alpha = 0.95$) superimposed over a heatmap of the 2DEH is shown below:

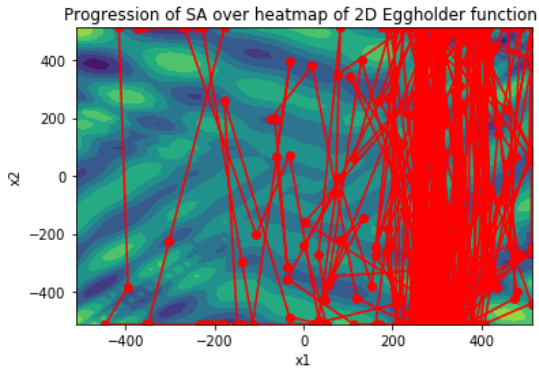

Figure 2.2: Path between solutions generated by SA superimposed over heatmap of the 2DEH

It can be seen that the SA manages to explore a large portion of the entire search space and manages to arrive at very close to the global minimum. A scatter plot of the points on the search space explored for a case where the SA arrived atthe global minimum (-959.6326, x1, x2 = 512.0, 404.3158, parameters:($k = 20$, $L_k = 500$, $\alpha = 0.95$) superimposed over a heatmap of the 2DEH is shown below:

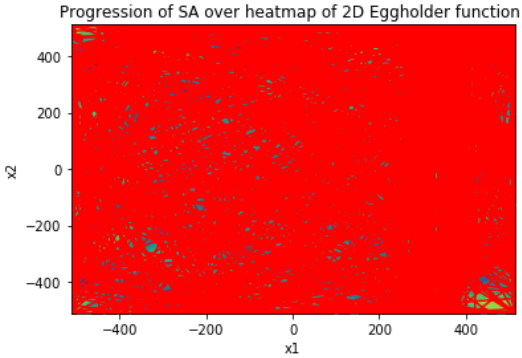

Figure 2.3: Path between solutions generated by SA superimposed over heatmap of the 2DEH

It can be seen in this case that the SA explores the entire search space.To investigate the reliability of the SA for the 2DEH, the algorithm was repeated 50 times and a histogram of the solutions found is given below:

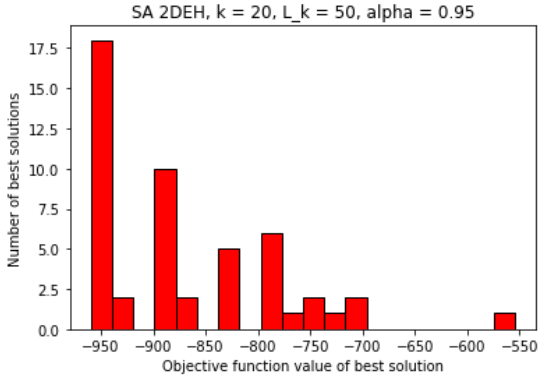

Figure 2.4: Histogram of solutions obtained by implementing SA on 2DEH 50 times.

The algorithm attains solutions very close to the global minimum (i.e. in the region of -950) around one third of the time. This is substantially less than ideal but suggests that it might sometimes yield good results on the 5DEH.

2.2 5D Eggholder function

The performance of SA on the 5DEH was investigated by plotting histograms of 50 SA applications to the 5DEH for given parameter settings. The following experiments are not meant to be exhaustive but to elucidate some rough relationships between parameter settings and algorithmic performance.

The first two experiments set $\alpha_k = 0.95$ and investigated two different configurations of k and L_k (20,500 & 100,100). The best result, mean of the results and standard deviation are computed. The second two experiments involved decrementing T according to the Huang method described halfway down page 7 of the lecture notes [1]

For the case where $\alpha_k = 0.95$:

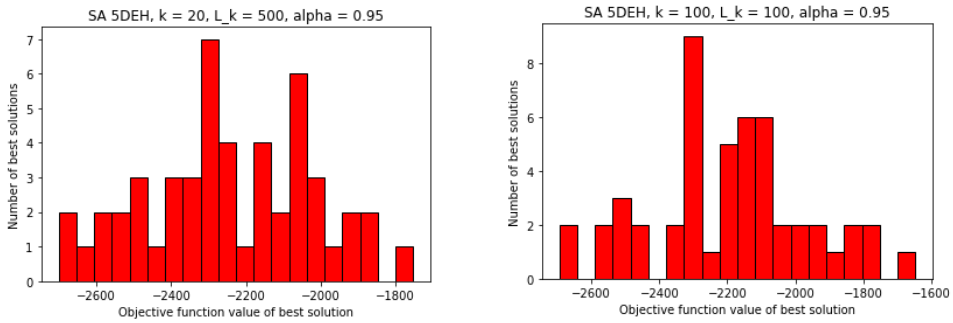

Figure 2.5: Histograms of solutions found by 50 implementations of SA on 5DEH for different parameter settings with $\alpha_k = 0.95$

$\alpha_k = 0.95$

	Best result	Mean	Std deviation	Co-ordinates
k=20,lk=500	-2700.55	-2243.58	222.21	(512.0, 399.78, 512.0, -276.54, 512.0)
k=100,lk=100	-2694.53	-2191.84	229.11	(512.0, 393.33, 512.0, 391.70, 512.0)

For the case where α_k is decremented according to the Huang schedule:

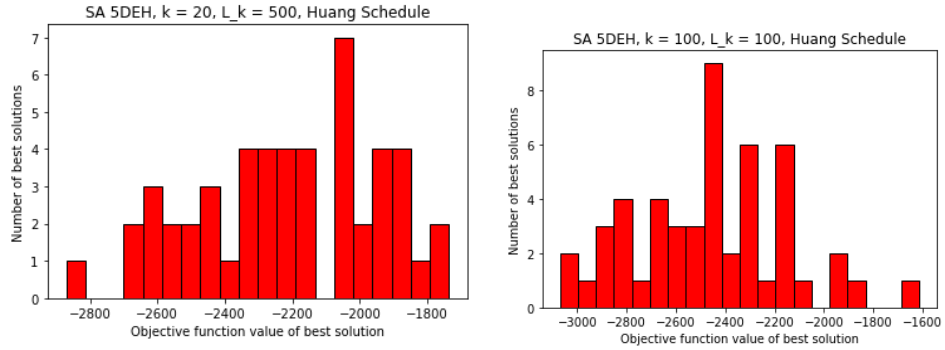

Figure 2.6: Histograms of solutions found by 50 implementations of SA on 5DEH for different parameter settings with α_k decremented according to Huang's method

Huang Schedule

	Best result	Mean	Std deviation	Co-ordinates
k=20,lk=500	-2870.17	-2216.30	269.35	(362.15, 512.0, 387.0, -354.42, 512.0)
k=100,lk=100	-3069.62	-2455.56	311.50	(-339.74, 512.0, -279.58, 512.0, 397.46)

Using the Huang method appears to yield slightly better best solutions, although in this case this is very possibly due to chance. The standard deviation between solutions of different implementations seems to be higher in the case of the Huang method.

Chapter 3

Genetic Algorithm

The GA algorithm was built according to the basic structure given in the lecture notes [2]. Solutions were encoded as 15 bit strings, giving quantization errors of $512/2^15 = 0.015625$. Parent selection was done using the method attributed to Baker[1985] at the bottom of page 3 of the lecture notes. The initial population was generated randomly.

There were several parameters of interest for GAs. These included the mutation probability (P_m), the cross-breeding probability (P_c), the size of a generation and the number of generations to iterate over. As the number of objective function evaluations for an application of a given algorithm was limited to 10 000 for this assignment, these last two parameters were constrained according to (generation size) = 10000/(number of generations). Hence the number of parents was chosen as the independent variable to alter.

3.1 2D Eggholder Function

The algorithm was first tested on the Eggholder in 2 dimensions.

A scatter plot of the co-ordinates of the best solution in each generation for a case where the GA performed reasonably well (best solution = -952.09) is superimposed over the heatmap of the 2DEH is given below:

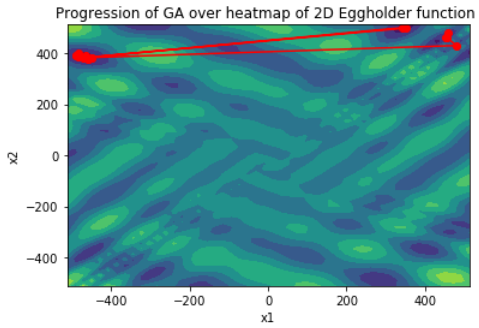

Figure 3.1: Path between best solutions at each generation superimposed over heatmap of 2DEH

A version of this plot magnified over the region explored by the GA is given below, with the global minimum given as a yellow dot:

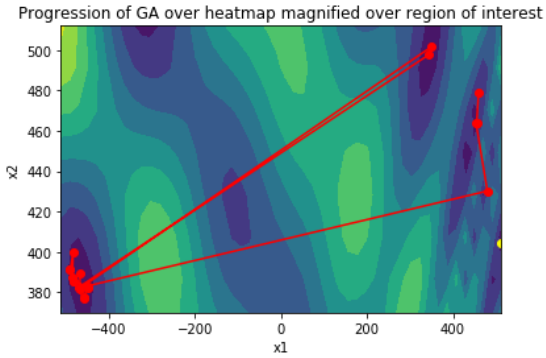

Figure 3.2: Magnified path between best solutions at each generation superimposed over heatmap of 2DEH

While the GA moves around near the region of the global minimum, it does not reach it. While a solution relatively close in objective function value to the global minimum is found, only a small part of the search space is explored. It appears that the GA has more trouble than SA in exploring the search space, at least in the case of two dimensions.

It can be seen that the GA manages to explore a large portion of the entire search space and manages to arrive at very close to the global minimum. To investigate the reliability of the SA for the 2DEH, the algorithm was repeated 50 times and a histogram of the solutions found is given below:

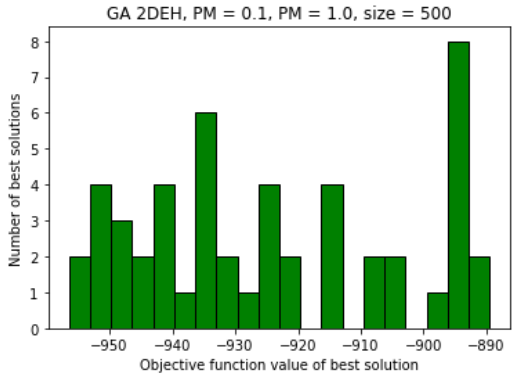

Figure 3.3: Histogram of solutions obtained by implementing SA on 2DEH 50 times.

The GA appears to perform worse than the SA when applied to the 2DEH. It was already noted above how the GA appears to explore substantially less of the search space when compared to SA. Only a small range of parameter settings was investigated for the 2DEH so it is possible that a different configuration of these settings would lead to a stronger performance. While it would have been preferred to spend more time examining the code and structure of the GA in order to determine whether or not either of these were at fault, due to time constraints it was decided to go ahead and see if the GA could yield any useful results when applied to the 5DEH.

3.2 5D Eggholder Function

The experiments were performed at two different values of population size (100 & 500). The experiments investigated the performance for two configurations of P_m and P_c (0.1,1.0 & 0.01,0.5).

For the case where size = 500:

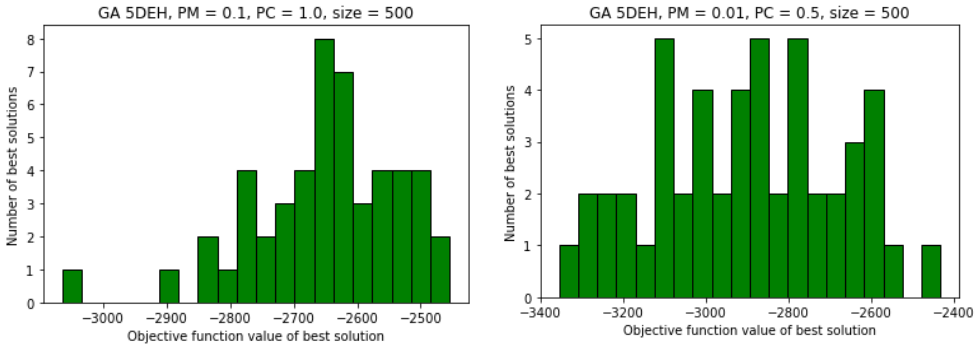

Figure 3.4: Histograms of solutions found by 50 implementations of SA on 5DEH for different parameter settings where the generation size is 500

Population size = 500, number of generations = 20

	Best result	Mean	Std deviation	Co-ordinates
Pm=0.1,Pc=1.0	-3064.95	-2650.10	117.85	(466.71, 388.29, -371.11, -445.54, 387.17)
Pm=0.01,Pc=0.5	-3354.20	-2905.42	219.85	(463.18, 420.24, 442.73, 459.06, -300.0)

For the case where size = 100:

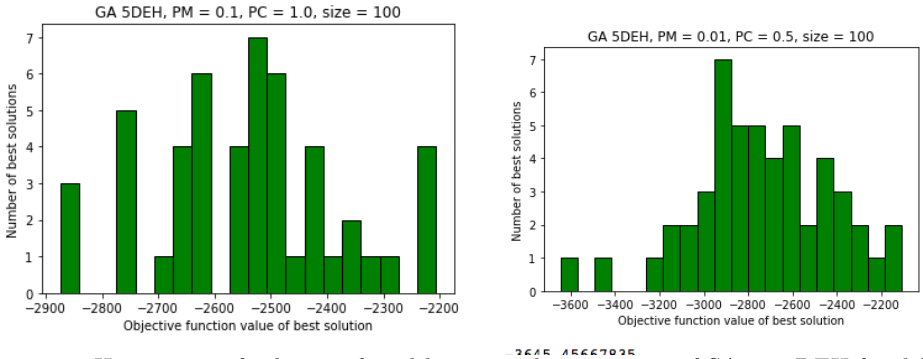

Figure 3.5: Histograms of solutions found by 50 implementations of SA on 5DEH for different parameter settings where the generation size is 100

Population size = 100, number of generations = 100

	Best result	Mean	Std deviation	Co-ordinates
Pm=0.1,Pc=1.0	-2874.25	-2542.18	164.72	(469.12, -298.69, -461.97, -394.29, -418.05)
Pm=0.01,Pc=0.5	-3645.46	-2741.67	315.35	(343.50, 491.42, 441.67, 453.78, 466.40)

There doesn't appear to be a significant difference in how the GA performs regarding population size, although better final solutions and higher means were found in both cases where $P_m = 0.01, P_c = 0.5$. It appears that being too eager to allow the populations to mutate and breed means that very good characteristics might be immediately flushed out of the populations. It is clear that the trade-off between obtaining maximum variation and maintaining good traits in the population needs to be delicately balanced.

Chapter 4

Comparison of algorithms

The SA algorithm performed substanitally better than the GA algorithm when applied to the 2DEH, often arriving at or in the neighborhood of the global minimum (or achieving function values close to the minimum even if the points are far away from the true minimum). When applied to the 5DEH, however, the GA generally got better results, with the mean best result of a sequence of 50 applications of GA being often lying closer to -3000 than while the equivalent results for the SA tended to lie closer to -2000. Due to the inability to visually examine the search space of the 5DEH, it is more challenging to examine how either algorithm searches this space compared to the case of the 2DEH. It may be the case that it is not possible to extrapolate the behaviour of both algorithms in 2D to a 5D context. If so, it may be the case that the GA was much better able to explore the 5D space.

The maps of how each algorithm explored the search space in 2D might also be deceptive, as in the case of SA every accepted solution was plotted, and only one solution per generation for GA was plotted. This doesn't capture the extent to which GA allows for diversity among its members, and such diversity may gain in strength with increasing dimensions relative to the SA method of generating new solutions.

The minimum value achieved for the 5DEH, over all experiments, was -3645.457, at the co-ordinates (343.5039, 491.4182, 441.6658, 453.7838, 466.4014).

Appendix A

Code

A.1 Simulated_Annealing.py

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(44)

##############################
#### Objective function defined
##############################

def obj(x1, x2, x3, x4, x5):
    A = -(x2+47)*np.sin(np.sqrt(np.abs(x2+0.5*x1+47))) - x1*np.sin(np.sqrt(np.
        abs(x1 - x2 - 47)))
    B = -(x3+47)*np.sin(np.sqrt(np.abs(x3+0.5*x2+47))) - x2*np.sin(np.sqrt(np.
        abs(x2 - x3 - 47)))
    C = -(x4+47)*np.sin(np.sqrt(np.abs(x4+0.5*x3+47))) - x3*np.sin(np.sqrt(np.
        abs(x3 - x4 - 47)))
    D = -(x5+47)*np.sin(np.sqrt(np.abs(x5+0.5*x4+47))) - x4*np.sin(np.sqrt(np.
        abs(x4 - x5 - 47)))
    return A + B + C + D

def map(A):
    return 1024*A - 512

##########################
##########
##########################

############################
########## function max_min() clips any numbers that go outside range
    -512->512
##########################


def max_min(A):
    if A > 1:
        return 1
```

```
    if A < 0:
        return 0
    else:
        return A

############################
########## function update(generates new solutions (current))
############################


def update(current, D, T):

    alpha = 0.1
    omega = 2.1


    D_new = np.zeros((5, 5))
    next = np.zeros(5)

    R = np.array([D[0][0]*(2*np.random.rand() - 1), D[1][1]* (2*np.random.rand
        () - 1), D[2][2]* (2*np.random.rand() - 1),
                  D[3][3]*(2*np.random.rand() - 1), D[4][4]* (2*np.random.rand
                      () - 1)])

    next[0] = max_min(current[0] + R[0])
    next[1] = max_min(current[1] + R[1])
    next[2] = max_min(current[2] + R[2])
    next[3] = max_min(current[3] + R[3])
    next[4] = max_min(current[4] + R[4])

    delta = obj(map(next[0]), map(next[1]), map(next[2]), map(next[3]), map(
        next[4])) - obj(map(current[0]),
                 map(current[1]), map(current[2]), map(current[3]), map(current
                     [4]))


    d = np.sqrt(R[0]**2 + R[1]**2 + R[2]**2 + R[3]**2 + R[4]**2)

    R = np.fabs(R)

    D_new[0][0] = (1-alpha)*D[0][0] + alpha*omega*abs(R[0])
    D_new[1][1] = (1-alpha)*D[1][1] + alpha*omega*abs(R[1])
    D_new[2][2] = (1-alpha)*D[2][2] + alpha*omega*abs(R[2])
    D_new[3][3] = (1-alpha)*D[3][3] + alpha*omega*abs(R[3])
    D_new[4][4] = (1-alpha)*D[4][4] + alpha*omega*abs(R[4])

    if delta < 0:
        return next, D_new

    elif np.exp(-delta/(T*d)) > np.random.rand():
        return next, D_new

    else:
        return current, D
```

```
############################
########## function init_T() initialises the temperature
############################

def init_T(current, D, n):

    alpha = 0.1
    omega = 2.1

    obj_list = []

    for i in range(0, n):

        D_new = np.zeros((5, 5))
        next = np.zeros(5)

        R = np.array([D[0][0]*(2*np.random.rand() - 1), D[1][1]* (2*np.random.
            rand() - 1), D[2][2]* (2*np.random.rand() - 1),
                      D[3][3]*(2*np.random.rand() - 1), D[4][4]* (2*np.random.
                          rand() - 1)])

        next[0] = max_min(current[0] + R[0])
        next[1] = max_min(current[1] + R[1])
        next[2] = max_min(current[2] + R[2])
        next[3] = max_min(current[3] + R[3])
        next[4] = max_min(current[4] + R[4])

        R = np.fabs(R)

        D_new[0][0] = (1-alpha)*D[0][0] + alpha*omega*abs(R[0])
        D_new[1][1] = (1-alpha)*D[1][1] + alpha*omega*abs(R[1])
        D_new[2][2] = (1-alpha)*D[2][2] + alpha*omega*abs(R[2])
        D_new[3][3] = (1-alpha)*D[3][3] + alpha*omega*abs(R[3])
        D_new[4][4] = (1-alpha)*D[4][4] + alpha*omega*abs(R[4])

        f = obj(map(next[0]), map(next[1]), map(next[2]), map(next[3]), map(
            next[4]))
        obj_list.append(f)

        current = next
        D = D_new

    T = np.std(obj_list)
    return T


############################
########## function main() implements SA on objective function
############################

def main(k, L_k):

    res_list = []
```

```
    solns = []

    ###### Initial solution generated randomly

    init = np.array([np.random.rand(), np.random.rand(), np.random.rand(), np.
        random.rand(), np.random.rand()])

    D_start = np.zeros((5,5))

    for i in range(0, 5):
        D_start[i][i] = init[i]

    T = init_T(init, D_start, 100)

    current, D = update(init, D_start, T)

    for j in range(0, k):
        eta_list = []
        for i in range(0, L_k):

            X, Y = update(current, D, T)
            if np.any(current != X) == True:
                current, D = X, Y
                eta_list.append(obj(map(current[0]), map(current[1]), map(
                    current[2]), map(current[3]), map(current[4])))
                res_list.append(obj(map(current[0]), map(current[1]), map(
                    current[2]), map(current[3]), map(current[4])))
                solns.append((map(current[0]), map(current[1]), map(current
                    [2]), map(current[3]), map(current[4])))

        alpha_2 = max(0.5, np.exp(-0.7*T/np.std(eta_list)))
        #alpha_2 = 0.95
        T = alpha_2*T

    best = min(res_list)
    best_id = res_list.index(best)
    co_ords = solns[best_id]

    return best, co_ords, res_list, solns



##################################
#### Code below gives example of SA being run on obj function 50 times, with
    results plotted in a histogram
##################################

ult_res = []
ult_solns = []

kk = 20
L_kk = 500

for w in range(0, 50):
```

```
    a = main(kk, L_kk)
    ult_res.append(a[0])
    ult_solns.append(a[1])

ult_best = min(ult_res)
ult_best_id = ult_res.index(ult_best)
ult_soln = ult_solns[ult_best_id]
print ult_best, ult_soln


plt.hist(ult_res, 20, color='red', histtype='bar', ec='black')
plt.title('SA 5DEH, k = 20, L_k = 500, Huang Schedule')
plt.xlabel('Objective function value of best solution')
plt.ylabel('Number of best solutions')
plt.show()

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

A.2 Genetic_Algorithm.py

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(44)

##############################
#### Objective function defined
##############################


def obj(x1, x2, x3, x4, x5):
    A = -(x2+47)*np.sin(np.sqrt(np.abs(x2+0.5*x1+47))) - x1*np.sin(np.sqrt(np.
        abs(x1 - x2 - 47)))
    B = -(x3+47)*np.sin(np.sqrt(np.abs(x3+0.5*x2+47))) - x2*np.sin(np.sqrt(np.
        abs(x2 - x3 - 47)))
    C = -(x4+47)*np.sin(np.sqrt(np.abs(x4+0.5*x3+47))) - x3*np.sin(np.sqrt(np.
        abs(x3 - x4 - 47)))
    D = -(x5+47)*np.sin(np.sqrt(np.abs(x5+0.5*x4+47))) - x4*np.sin(np.sqrt(np.
        abs(x4 - x5 - 47)))
    return A + B + C + D

##############################
##### Function which maps range 0->32767 to -512->512, and clips numbers that
    go beyond this boundary
##############################

def map(A):
    B = 1024.0*A/32767 - 512
    if B > 512:
```

```
        return 512
    elif B < -512:
        return -512
    else:
        return B



############################
########## function mate() that breeds two parents and replaces them with
    their offspring
############################

def mate(parA, parB):
    parA = '{0:b}'.format(parA) ########Parent numbers converted into binary#
    parB = '{0:b}'.format(parB)

    L_s = min(len(parA), len(parB))

    clipA = parA[:-L_s]    #####Extra digits "clipped" so crossover happens
        between strings of equivalent length
    clipB = parB[:-L_s]

    rsA = parA[-L_s:]
    rsB = parB[-L_s:]

    nz = [int(rsA[i]) - int(rsB[i]) for i in range(L_s)] ###### crossover is
        performed on reduced surrogates
    inds = [i for i, e in enumerate(nz) if e != 0]

    try:
        beg = min(inds)
        end = max(inds)
    except:
        return [int("".join(parA), 2), int(''.join(parB), 2)]

    RSA = rsA[beg:end+1]
    RSB = rsB[beg:end+1]

    LL_s = min(len(RSA), len(RSB))
    LL_b = min(len(RSA), len(RSB))

    prob = [1.0/2**(LL_s-1-i) for i in range(LL_s)]   ####The further to the
        left the binary digit, the smaller the
                                                      ##### probability of the
                                                          break point being
                                                        located there
    for i in prob:
        if np.random.rand() < i:
            place = prob.index(i)
            break

    br = LL_s - place
```

```
    chA = RSA[:-br] + RSB[-br:]
    chB = RSB[:-br] + RSA[-br:]

    rsA = clipA + rsA[0:beg] + chA + rsA[end+1:]
    rsB = clipB + rsB[0:beg] + chB + rsB[end+1:]

    return [int("".join(rsA), 2), int(''.join(rsB), 2)]    #### Numbers
        converted back from binary

##########################
########## function mut() which mutates input numbers
##########################


def mut(parents, rate):
    parentA = '{0:b}'.format(parents[0])    ######## Converted into binary
    parentB = '{0:b}'.format(parents[1])
    parentC = '{0:b}'.format(parents[2])
    parentD = '{0:b}'.format(parents[3])
    parentE = '{0:b}'.format(parents[4])

    midA = [(ch if np.random.rand() >= rate else str(1-int(ch))) for ch in
        parentA]  ###Each bit has same probability
    midB = [(ch if np.random.rand() >= rate else str(1-int(ch))) for ch in
        parentB] ##### of flipping
    midC = [(ch if np.random.rand() >= rate else str(1-int(ch))) for ch in
        parentC]
    midD = [(ch if np.random.rand() >= rate else str(1-int(ch))) for ch in
        parentD]
    midE = [(ch if np.random.rand() >= rate else str(1-int(ch))) for ch in
        parentE]
    return [int("".join(midA),2), int("".join(midB),2), int("".join(midC),2),
        int("".join(midD),2), int("".join(midE),2)]



##########################
########## function next_gen() builds next generation out of best performing
    solutions in current generation
##########################

def next_gen(current):

    next = []

    fitness = [obj(map(x[0]), map(x[1]), map(x[2]), map(x[3]), map(x[4])) for
        x in current] ##########Solutions are ranked
    ranking = [fitness.index(x) for x in np.sort(fitness)]
    current = [current[x] for x in ranking]



    N = len(fitness)
    S = 2.0
```

```
    selection = [((S*(N+1-2*x)+ 2*(x-1))/(N-1)) for x in range(1, N+1)]
    rem = selection - np.floor(selection)

    for i in range(N):
        for j in range(int(np.floor(selection)[i])):
            next.append(current[i])

    while len(next) < N:
        for i in range(N):
            if rem[i] > np.random.random_sample() and len(next) < N :
                next.append(current[i])

    return next


############################
########## function pair_mate() randomly selects parents and cross breeds
    them with probability selected by "prob" argument
############################


def pair_mate(gen, prob):
    np.random.shuffle(gen)
    h1 = gen[:len(gen)/2]
    h2 = gen[len(gen)/2:]

    for i in range(len(gen)/2):
        if np.random.random_sample() < prob:
            h1[i][0], h2[i][0] = mate(h1[i][0], h2[i][0])
            h1[i][1], h2[i][1] = mate(h1[i][1], h2[i][1])
            h1[i][2], h2[i][2] = mate(h1[i][2], h2[i][2])
            h1[i][3], h2[i][3] = mate(h1[i][3], h2[i][3])
            h1[i][4], h2[i][4] = mate(h1[i][4], h2[i][4])

    return h1 + h2

############################
########## function iter() takes as input the current generation probability
    of recombination and mutation probability
########## and calls the above functions to produce a new generation of
    solutions
############################


def iter(current, prob, rate):
    next = next_gen(current)
    next = pair_mate(next, prob)
    next = [mut(next[i], rate) for i in range(len(next))]

    return next

############################
########## function main() takes as inputs the probability of recombination,
    the mutation probability and the size of
```

```
########## generation and runs the GA on the 5DEH function for 10 000
    objective function evaluations.
##########################


def main(PM, PC, parents):

    num_it = 10000/parents
    Pm = PM
    Pc = PC

    res = []
    res_av = []
    X_pts = []
    Y_pts = []
    solns = []


    next = []                       ###### Members of first generation are
        randomly selected
    for i in range(0, parents):
        next.append([np.random.randint(0, 32767), np.random.randint(0, 32767),
                 np.random.randint(0, 32767), np.random.randint(0, 32767), np.
                     random.randint(0, 32767)])



    for i in range(0, num_it):
        cur = [obj(map(x[0]), map(x[1]), map(x[2]), map(x[3]), map(x[4])) for
            x in next]
        solns.append((map(next[cur.index(min(cur))][0]), map(next[cur.index(
            min(cur))][1]), map(next[cur.index(min(cur))][2]),
                      map(next[cur.index(min(cur))][3]), map(next[cur.index(
                          min(cur))][4])))
        res.append(min(cur))
        res_av.append(np.mean(cur))
        next = iter(next, Pc, Pm)

    A_pts = [x[0] for x in solns]
    B_pts = [x[1] for x in solns]
    C_pts = [x[2] for x in solns]
    D_pts = [x[3] for x in solns]
    E_pts = [x[4] for x in solns]

    pts = [A_pts, B_pts, C_pts, D_pts, E_pts]


    best = min(res)
    co_ords = solns[res.index(best)]

    return res, res_av, solns, best, co_ords, pts

###################################
```

```
#### Code below gives example of GA being run on obj function 50 times, with
    results plotted in a histogram
##################################

ult_res = []
ult_solns = []

PPm = 0.01
PPc = 0.5

size = 500

for w in range(0, 50):
    a = main(PPm, PPc, size)
    ult_res.append(a[3])
    ult_solns.append(a[4])



ult_best = min(ult_res)
ult_best_id = ult_res.index(ult_best)
ult_soln = ult_solns[ult_best_id]
print ult_best, ult_soln

plt.hist(ult_res, 20, color = 'green', histtype='bar', ec='black')
plt.title('GA 5DEH, PM = 0.01, PC = 0.5, size = 500')
plt.xlabel('Objective function value of best solution')
plt.ylabel('Number of best solutions')
plt.show()

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

Bibliography

[1] Vle.cam.ac.uk. (2018). Moodle: Log in to the site. [online] Available at: https://www.vle.cam.ac.uk/pluginfile.php/11351401/mod_resource/content/1/4M17SimulatedAnnealing.pdf [Accessed 16 Jan. 2018].

[2] Vle.cam.ac.uk. (2018). Moodle: Log in to the site. [online] Available at: https://www.vle.cam.ac.uk/pluginfile.php/1735021/mod_resource/content/2/4M17GeneticAlgorithms.pdf [Accessed 16 Jan. 2018].