Metaheuristics Assignment

Exercise 1

(a) Formulate the revenue maximization problem for this flight as an optimization problem.

The maximization problem was converted to a minimization one (see below).

$$D_i = a_i \exp\left(-\frac{1}{a_i} p_i\right)$$

$a_1 = 100$
$a_2 = 150$
$a_3 = 300$

Limiting factor no. of seats = 150

3 prices: p_1, p_2, p_3

Revenue function: $J(p_1, p_2, p_3) = (p_1 D_1) + (p_2 D_2) + (p_3 D_3)$

Maximise function under equality constraint:

$$\sum_{i=1}^{3} a_i e^{-\frac{p_i}{a_i}} - 150 = 0$$

Minimise the negative of the function:

$$\min_{p} \; -J \quad \text{s.t. } g = 0,\ p_i >= 0$$

Using the Lagrangian:

$$L = -J + \lambda g$$

$$p_i = a_i + \lambda$$

(b) What are the optimal prices and how many people are expected to buy a ticket in each fare bucket?

The calculations were done in the Python script *Ex1_SensitivityAnalysis.py* using the scipy.optimize.fsolve() functionality. After solving for lambda, the price buckets (p_1, p_2, p_3) could be calculated which then meant the passenger buckets (D_1, D_2, D_3) could also be calculated. Once the price and passenger buckets were calculated the revenue could then be calculated.

The output of the Python script is the following:

```
The total number of seats is:  150
Lambda is:  [56.74841953]
Price buckets :- p1 is:[156.74841953], p2 is:[206.74841953], p3 is:[356.74841953]
No. of passengers :- D1 is:20.86, D2 is:37.8, D3 is:91.34
Revenue is:  [43670.85883515]
```

I am assuming that the price is in $ as the flight is from Boston to Atlanta so probably an American airline. The prices for each p bucket are the following (rounded to 2 dcp):

p_1 = $156.75 p_2 = $206.75 p_3 = $356.75

The value of lambda (the Lagrangian multiplier) was found to be 56.7484.

The number of passengers for each passenger bucket (after rounding) is the following:

D_1 = 21 D_2 = 38 D_3 = 91

21 + 38 + 91 = 150 people

The total revenue for a full flight was calculated to be **$43670.86** (rounded to 2dcp)

(c) Using sensitivity analysis, if the airline were to squeeze three additional seats onto this flight,
a. How much do you expect revenue to change?
b. By how much should the airline change each price?

The Python script was run again this time setting the number of seats variable to 153.

Below is the output of the script:

```
The total number of seats is:  153
Lambda is:  [52.87443249]
Price buckets :- p1 is:[152.87443249], p2 is:[202.87443249], p3 is:[352.87443249]
No. of passengers :- D1 is:21.68, D2 is:38.79, D3 is:92.53
Revenue is:  [43835.26961528]
```

The new revenue amount is **$43835.27** which is an increase of:

$43835.27 - $43670.86 = **$164.41**

The new price buckets (rounded to 2dcp) are the following:

p_1 = $152.87 p_2 = $202.87 p_3 = $352.87

The change for each price bucket is the following:

p_1 = $156.75 - $152.87 = $3.88 p_2 = $206.75 - $202.87 = $3.88 p_3 = $356.75 - $352.87 = $3.88

Which means that the airline should reduce each price by: **$3.88**.

The new passenger buckets have the following values:

D_1 = 22 D_2 = 39 D_3 = 92

22 + 39 + 92 = **153**

Exercise 2

Ex. 2 Part 1 – Minimisation of the Rosenbrock function

In order to minimise the Rosenbrock function:

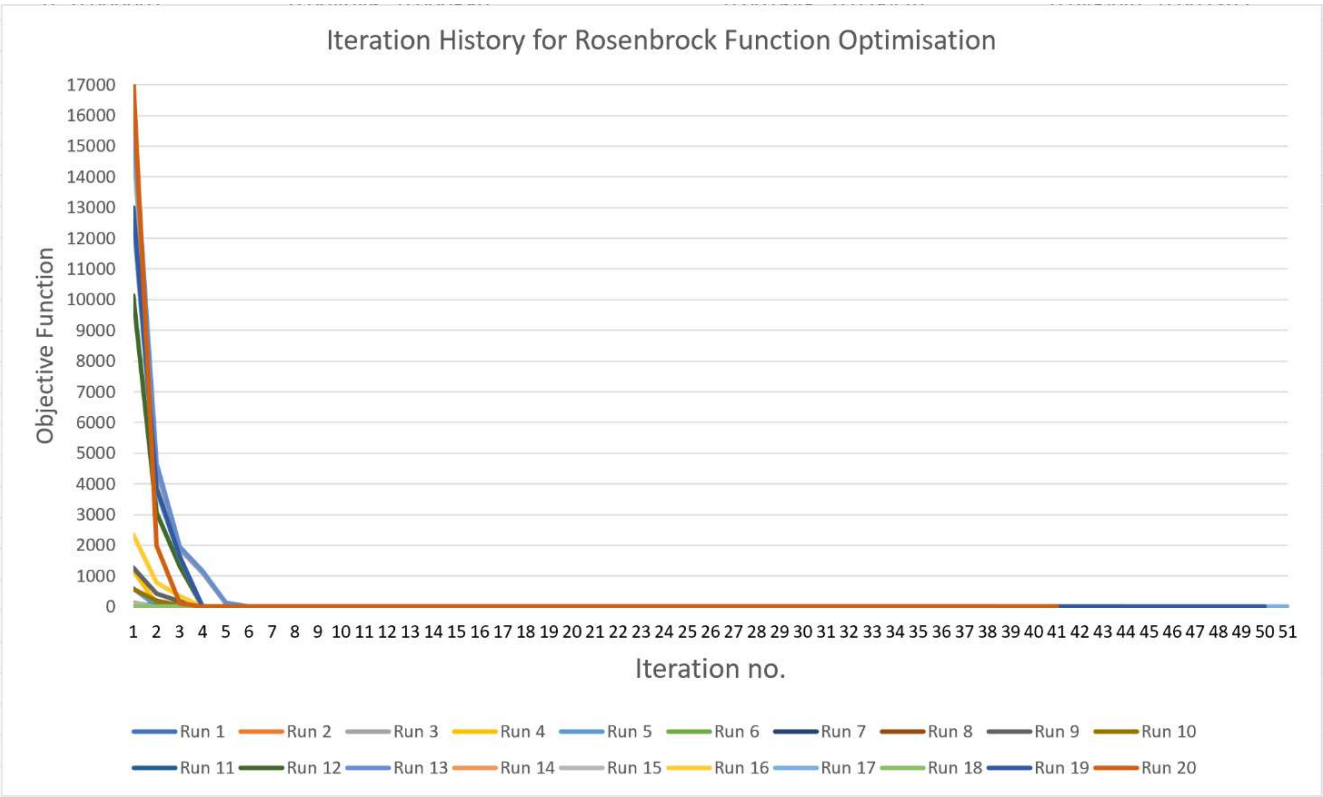
$$\text{Minimize } f(\mathbf{x}) = 100\left(x_2 - {x_1}^2\right)^2 + (1 - x_1)^2$$

the gradient based method of choice was Python's scipy Newton method. This was done by setting 'method' to Newton-CG in scipy.optimize.minimize(). The CG refers to the fact that an internal inversion of the Hessian is performed by conjugate gradient. As the method performs better if the gradient and Hessian of the function are passed directly to the minimize function, these were calculated in separate functions in the Python script (Exercise2_Rosenbrock.py) and the function names were passed directly to the 'jac' and 'hess' parameters in scipy.optimize.minimize() respectively. The values for the x parameters were generated randomly within a lower and upper range of [-5,5] using Python's random.uniform() functionality. For simplicity's sake the same values for x1 and x2 were used per run. 20 runs were done in total.

As specified in the assignment paper, the Rosenbrock function has a known minimum at [1,1] with an optimal function of value 0. In all 20 runs the algorithm was able to find the global minimum and the optimal objective function was always zero. The minimum number of iterations was 8, the maximum was 51.

The Python script used was the following: *Ex2_Rosenbrock.py*

Below is a graph showing the iteration history for the minimisation of the Rosenbrock function. It shows that most runs are quite close to an objective function of zero already by iteration 4 and almost all by iteration 6 although some runs took longer getting to an actual zero value.



The optimisation information for all the runs for the Rosenbrock (Banana) function is summarised in the below table.

Run no.	Starting points	Final Solution points	Feasible (Y/N)	Objective Function Solution	No. of iterations
1	[2.17, 2.17]	[1.000000,1.000000]	Y	0.0000	26
2	[-0.12, -0.12]	[1.000000,1.000000]	Y	0.0000	30
3	[-4.53, -4.53]	[1.000000,1.000000]	Y	0.0000	38
4	[3.65, 3.65]	[1.000000,1.000000]	Y	0.0000	39
5	[3.22, 3.22]	[1.000000,1.000000]	Y	0.0000	33
6	[2.53, 2.53]	[1.000000,1.000000]	Y	0.0000	31
7	[-0.97, -0.97]	[1.000000,1.000000]	Y	0.0000	38
8	[1.23, 1.23]	[1.000000,1.000000]	Y	0.0000	13
9	[-2.17, -2.17]	[1.000000,1.000000]	Y	0.0000	32
10	[-1.7 -1.7]	[1.000000,1.000000]	Y	0.0000	36
11	[0.59 0.59]	[1.000000,1.000000]	Y	0.0000	24
12	[-4.04 -4.04]	[1.000000,1.000000]	Y	0.0000	44
13	[-4.6 -4.6]	[1.000000,1.000000]	Y	0.0000	38
14	[1.08 1.08]	[1.000000,1.000000]	Y	0.0000	12
15	[2.44 2.44]	[1.000000,1.000000]	Y	0.0000	28
16	[-2.62 -2.62]	[1.000000,1.000000]	Y	0.0000	44
17	[-4.31 -4.31]	[1.000000,1.000000]	Y	0.0000	51
18	[1.39 1.39]	[1.000000,1.000000]	Y	0.0000	21
19	[-4.34 -4.34]	[1.000000,1.000000]	Y	0.0000	50
20	[6.27 6.27]]	[1.000000,1.000000]	Y	0.0000	41

To summarise, it seems that the Newton-CG method was a very good one to use for minimising this particular function. However, it seems for functions with high dimensions it is not so suitable because the inversion of the Hessian can be very costly and unstable (large scale > 250).

Ex. 2 Part 2 – Minimisation of the Eggcrate function

In order to minimise the Eggcrate function:

$$\text{Minimize } f(\mathbf{x}) = x_1^2 + x_2^2 + 25\left(\sin^2 x_1 + \sin^2 x_2\right)$$

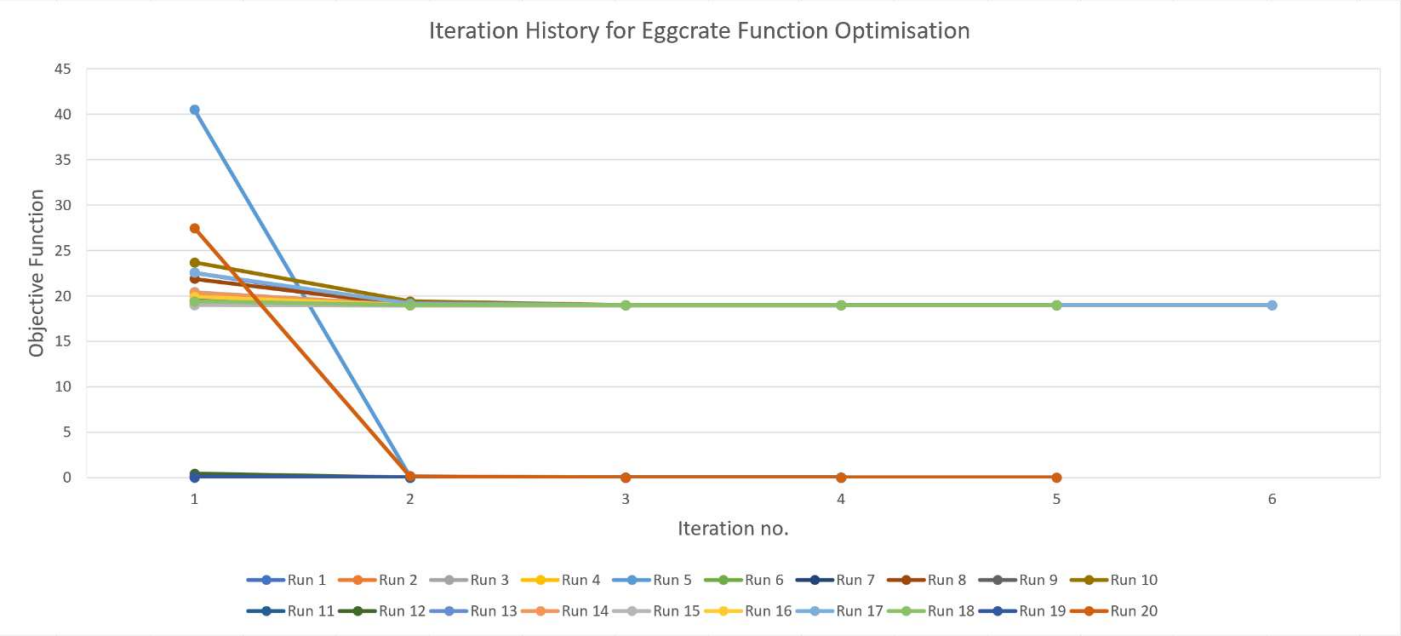
the same algorithm was used as for the Rosenbrock function: Newton-CG, as this is also an unconstrained problem and this algorithm is quite efficient with such problems. However, for the Eggcrate function the algorithm did not work as well as most of the time it got stuck at a local optimum. Again, the x values were generated randomly using the bounds $[-2\pi,2\pi]$ as defined in the assignment paper. The same x1 and x2 values were used per run. 20 runs were done in total.

Out of 20 runs only 6 were able to find the global minimum of [0, 0]. The objective function at the global minimum was also zero. Due to the shape of the function the local optimum was found at either [3.019602, 3.019602] or at [-3.019602, -3.019602]. The function value at these points was 18.9763. This means that there was only a 30% success rate compared with 100% with the Rosenbrock function.

The algorithm needed at most 6 iterations to find either a local or global optimum with 5 iterations being the most common. The least number of iterations was 3 in order find an optimum.

The Python script used was the following: *Ex2_Eggcrate.py*

Below is a graph showing the iteration history for the optimisation of the Eggcrate function.



From the above graph it can be seen that most of the runs converged to the local optimum of 18.9763.

The optimisation information for all of the runs for the Eggcrate function is summarised in the below table.

Run no.	Starting points	Final Solution points	Feasible (Y/N)	Objective Function Solution	No. of iterations
1	[2.33, 2.33]	[3.019602,3.019602]	Y	18.9764	5
2	[2.4, 2.4]	[3.019602,3.019602]	Y	18.9764	5
3	[2.64, 2.64]	[3.019602,3.019602]	Y	18.9764	5
4	[2.56 , 2.56]	[3.019602,3.019602]	Y	18.9764	5
5	[3.83, 3.83]	[0.000000,0.000000]	Y	0.0000	5
6	[3.48, 3.48]	[3.019602,3.019602]	Y	18.9764	5
7	[4.53, 4.53]	[3.019602,3.019602]	Y	18.9764	5
8	[2.16, 2.16]	[3.019602,3.019602]	Y	18.9764	5
9	[1.42, 1.42]	[0.000000,0.000000]	Y	0.0000	4
10	[-2.06, -2.06]	[-3.019602,-3.019602]	Y	18.9764	6
11	[1.62, 1.62]	[0.000000,0.000000]	Y	0.0000	3
12	[4.69, 4.69]	[0.000000,0.000000]	Y	0.0000	4
13	[-4.84, -4.84]	[-3.019602,-3.019602]	Y	18.9764	5
14	[3.55, 3.55]	[3.019602,3.019602]	Y	18.9764	4
15	[3.33, 3.33]	[3.019602,3.019602]	Y	18.9764	4
16	[4.62, 4.62]	[3.019602,3.019602]	Y	18.9764	5
17	[-3.61, -3.61]	[-3.019602,-3.019602]	Y	18.9764	6
18	[-2.7, -2.7]	[-3.019602,-3.019602]	Y	18.9764	5
19	[0.28, 0.28]	[0.000000,0.000000]	Y	0.0000	4
20	[3.84, 3.84]	[0.000000,0.000000]	Y	0.0000	5

To summarise it seems that those runs with starting points closer to the minimum such as 0.28, 1.62 and 1.42 always converged to the global minimum but some starting points such as 3.84, 3.83 and 4.69 which are not close to the global minimum also converged to zero. Therefore, I cannot see an obvious pattern as to when the algorithm converges and when it does not. The performance difference between the Rosenbrock minimisation and the Eggcrate is probably due to the eggcrate function being multi-modal. The Rosenbrock function only has one minimum whereas the eggcrate has multiple.

Ex. 2 Part 3 – Golinski's Speed Reducer Optimisation

The goal of this problem is to minimise the weight of the gearbox satisfying a number of constraints constituted by gear and shaft design. There are seven design variables listed in the table below with the lower and upper bounds specified.

Variable	Symbol	Units	Lower Bound (LB)	Upper Bound (UB)
Face width of gear face	x_1	cm	2.6	3.6
Teeth module	x_2	cm	0.7	0.8
Number of teeth of pinion	x_3	-	17	28
Length of shaft 1 between bearings	x_4	cm	7.3	8.3
Length of shaft 2 between bearings	x_5	cm	7.3	8.3
Diameter of shaft 1	x_6	cm	2.9	3.9
Diameter of shaft 2	x_7	cm	5.0	5.5

[1]

This problem has many constraints, 25 in total. There are 11 main constraints and the others are side constraints.

The 11 main constraints (below) need to be taken into consideration when carrying out the optimisation.

$27x_1^{-1}x_2^{-2}x_3^{-1} \leq 1$ G_1

$397.5x_1^{-1}x_2^{-2}x_3^{-2} \leq 1$ G_2

$1.93x_2^{-1}x_3^{-1}x_4^3x_6^{-4} \leq 1$ G_3

$1.93x_2^{-1}x_3^{-1}x_5^3x_7^{-4} \leq 1$ G_4

$[(745x_4x_2^{-1}x_3^{-1})^2 + 16.9x10^6]^{\frac{1}{2}}/[110.0x_6^3] \leq 1$ G_5

$[(745x_5x_2^{-1}x_3^{-1})^2 + 157.5x10^6]^{\frac{1}{2}}/[85.0x_7^3] \leq 1$ G_6

$x_2x_3/40 \leq 1.0$ G_7

$5x_2/x_1 \leq 1.0$ G_8

$x_1/12x_2 \leq 1.0$ G_9

$(1.5x_6 + 1.9)x_4^{-1} \leq 1.0$ G_{24}

$(1.1x_7 + 1.9)x_5^{-1} \leq 1.0$ G_{25} [1]

As this is a constrained problem, it was thought that the Sequential Quadratic Programming (SQP) method would be a suitable one for the optimisation. It was also mentioned in the assignment paper as providing a known feasible solution. SQP is an iterative method for non-linear optimisation.

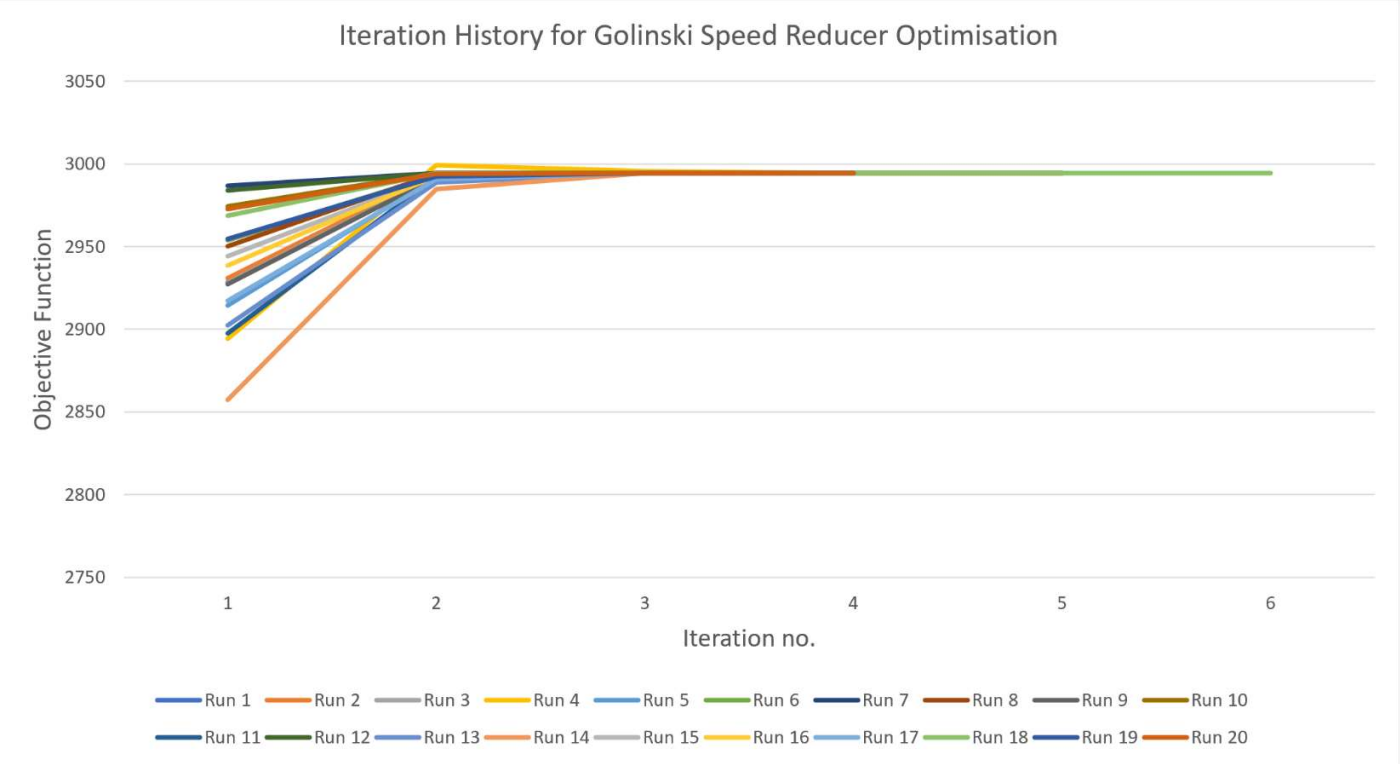
The function to minimise is the following:

$$f(x) = 0.7854x_1x_2^2(3.3333x_3^2 + 14.9334x_3 - 43.0934) - 1.5079x_1(x_6^2 + x_7^2) + 7.477(x_6^3 + x_7^3) + 0.7854(x_4x_6^2 + x_5x_7^2)$$

Once again, the minimisation was done via Python's scipy.optimize.minimize() functionality setting the 'method' parameter to SLSQP. The constraints were provided via the 'constraints' parameter as a NonLinearConstraint object. Only the 11 main constraints were used. Constraints G_{24} and G_{25} listed above were renamed as g10 and g11 in the Python optimisation script for convenience. The function takes 7 variables which were randomly generated again using Python's random.uniform() functionality. The values x1-x7 were generated according to the lower and upper bounds specified in the above table. There were 20 runs in total.

[1]https://www.researchgate.net/publication/329715526_A_Comparison_of_Sequential_Quadratic_Programming_Genetic_Algorithm_Simulated_Annealing_Particle_Swarm_Optimization_and_Hybrid_Algorithm_for_the_Design_and_Optimization_of_Golinski%27s_Speed_Reducer

The optimum points [3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867] were found for all runs. The optimum function value was found to be 2994 for every run with 2994.35 being the most common value (75% of the time) and only a very small variation of maximum 0.07 for the other runs. The smallest number of iterations to find the minimum was 3, the maximum was 6 with 4 being the most common. The graph below shows the iteration history for the optimisation of the Golinski problem.



From the graph it can be seen that all the runs converge to the same point after maximum 6 iterations.
The Python script used was the following: *Ex2_Golinski.py*

The table below summarises the results of the optimisation.

Run no.	Starting points	Final Solution points	Feasible (Y/N)	Objective Function Solution	No. of iterations
1	[2.928, 0.755, 21.228, 8.224, 7.957, 3.634, 5.195]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3545	4
2	[2.933,0.706, 24.534, 7.373, 8.177, 3.732, 5.235]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3661	5
3	[3.03, 0.773, 25.734, 7.382, 7.765, 2.998, 5.476]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3530	5
4	[2.634, 0.728, 25.734, 7.403, 7.486, 3.287, 5.423]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3599	5
5	[2.967, 0.795, 19.815, 7.93, 8.26, 3.074, 5.041]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3542	4
6	[3.208, 0.771, 18.224, 7.404, 7.74, 3.683, 5.172]	[3.5000, 0.7000, 17.0001, 7.3000, 7.7153, 3.3502, 5.2866]	Y	2994.3511	4
7	[3.581, 0.736, 25.133, 7.673, 7.318, 3.37, 5.464]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3988	4
8	[3.252, 0.793, 19.927, 8.262, 8.213, 3.541, 5.003]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3555	4
9	[3.014, 0.785, 24.781, 7.707, 8.165, 2.96, 5.226]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3549	4
10	[3.345, 0.781, 17.454, 8.209, 7.644, 3.526, 5.123]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2866]	Y	2994.3528	4
11	[2.818, 0.733, 27.19, 7.304, 7.832, 3.729, 5.489]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3547	5
12	[3.356, 0.727, 18.588, 8.121, 8.025, 3.184, 5.208]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3558	3
13	[3.008, 0.797, 27.005, 8.259, 7.705, 3.847, 5.17]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3584	5
14	[2.774, 0.787, 22.885, 7.869, 8.12, 3.857, 5.086]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3555	5
15	[3.152, 0.704, 19.072, 7.6, 8.256, 3.749, 5.092]	[3.4999, 0.7000, 17.0003, 7.3000, 7.7154, 3.3502, 5.2866]	Y	2994.3571	4
16	[3.184, 0.775, 21.381, 7.317, 8.222, 2.948, 5.493]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3580	4
17	[2.902, 0.744, 20.169, 7.83, 7.868, 3.745, 5.268]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3675	4
18	[3.183, 0.745, 27.251, 7.829, 8.087, 3.547, 5.254]	[3.4999, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.4283	6
19	[3.579, 0.705, 26.483, 8.172, 7.718, 3.753, 5.054]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3703	4
20	[3.214, 0.738, 18.137, 7.61, 7.705, 3.177, 5.387]	[3.5000, 0.7000, 17.0000, 7.3000, 7.7153, 3.3502, 5.2867]	Y	2994.3550	4

To summarise, it seems that Python's SLSQP minimise algorithm worked really well as the global minima for the seven variables were found with every run. There was a tiny variation with the objective function values which may possibly be solved by tweaking some of the minimize function options such as 'ftol' or 'eps' within the minimise function.

Exercise 3

Optimisation using Genetic Algorithm

As the title suggests the heuristic method chosen for this part was the Genetic Algorithm (GA). I decided to implement the genetic algorithm in order to have a better understanding of how they work. The genetic algorithm employed was written with the aid of the following youtube tutorials from Yarpiz.

https://www.youtube.com/watch?v=PhJgktRB1AM
https://www.youtube.com/watch?v=gIIygj3UlBs

The main algorithms for selection, crossover and mutation were kept the same as in the tutorial with adaptations made in the code only for the objective function, result printing and plotting of results.

The parents are selected using roulette wheel selection. Initially, random selection was used but roulette wheel selection gave better results. The crossover process is also random using Python numpy's random.uniform() functionality. The program uses Gaussian mutation using Python numpy's random.rand() for selecting positions.

For each of the minimization problems the process employed to tune the parameters were the following:

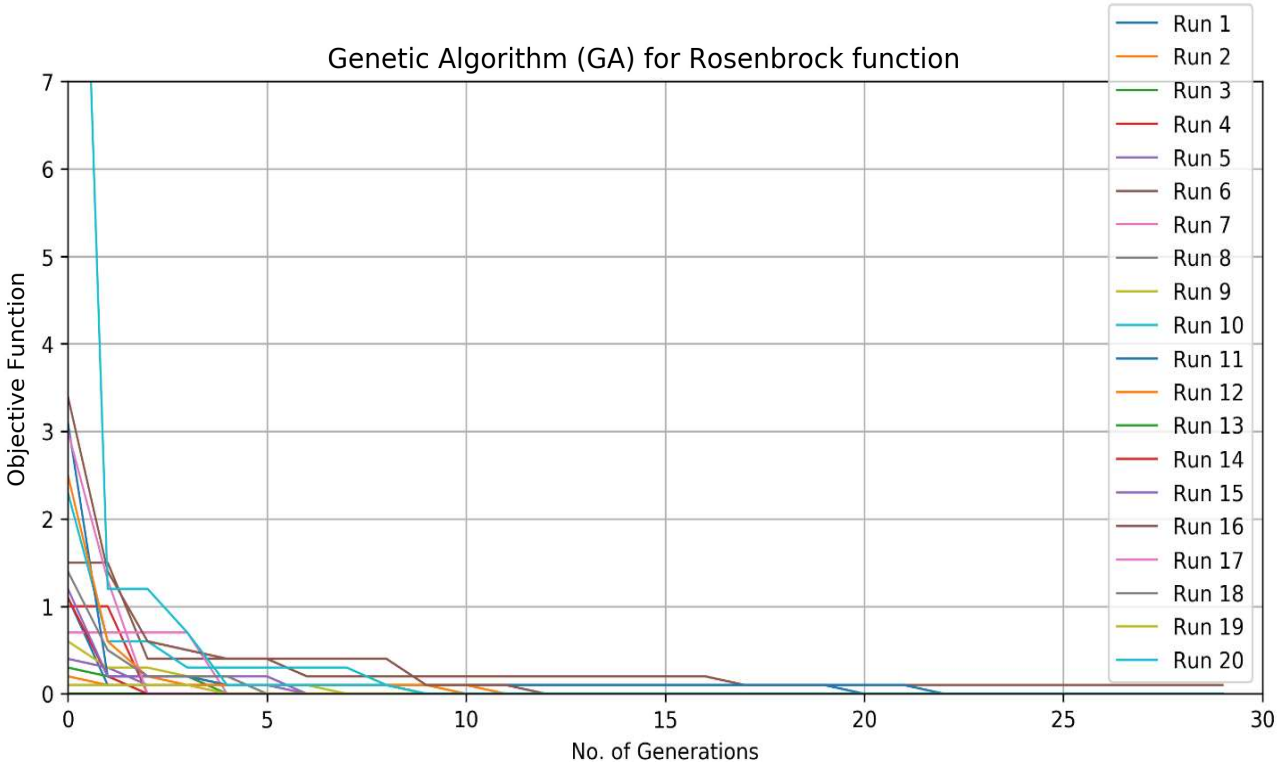
1) I first manipulated the population number and then I changed the maximum number of generations parameter. I continued to look at the interplay of these two parameters until I started getting some consistent results close to the optimal values.
2) Then I would start playing with the crossover parameter to see if tweaking it made a difference to the results.
3) Then I would change the mutation variables to see if changing them made any difference to the results.
4) Then I would run the genetic algorithm again and again to ensure that a more or less consistent result was achieved. If not, then I would start at 1) again.

Most of the time the greatest improvement was seen by either increasing the population number or number of generations. In general, the number of iterations was greater than the population number.

Ex. 3 Part 1 – Minimisation of the Rosenbrock function using GA

The following parameters were tuned in order to achieve the optimal results. There were 20 runs in total.

```
params.max_gen = 30 # Max no. of generations
params.pop_num = 60 # Population number
params.prpn_children = 1 # Proportion of children
params.gamma = 0.1 # Crossover parameter
params.mu = 0.53 # Mutation variable
params.sigma = 0.3 # Mutation variable
params.beta = 1 # Variable for roulette wheel selection
```



The graph shows that most of the runs came close to the optimal solution at around generation 12 but progress seems to be slow after that. Out of the 20 runs, the optimal objective function value of 0.0 was achieved in all cases bar one. The global minimum of [1, 1] was not reached in all cases but almost every run came very close to the optimum. The reason for not reaching the global minimum every time could be due to the fact that an elitism parameter was not used. With elitism the first or few best chromosomes are propagated to the new population. Without elitism such individuals may be lost if they are not selected to reproduce or if crossover or mutation eliminates them. This process significantly improves the GA's performance but alas was not employed in the genetic algorithm Python script for the Rosenbrock minimisation. A summary of the results for each run is given below.

The Python scripts used were the following: *rosen_app.py, rosen_ga.py*

Rosenbrock GA Results Summary

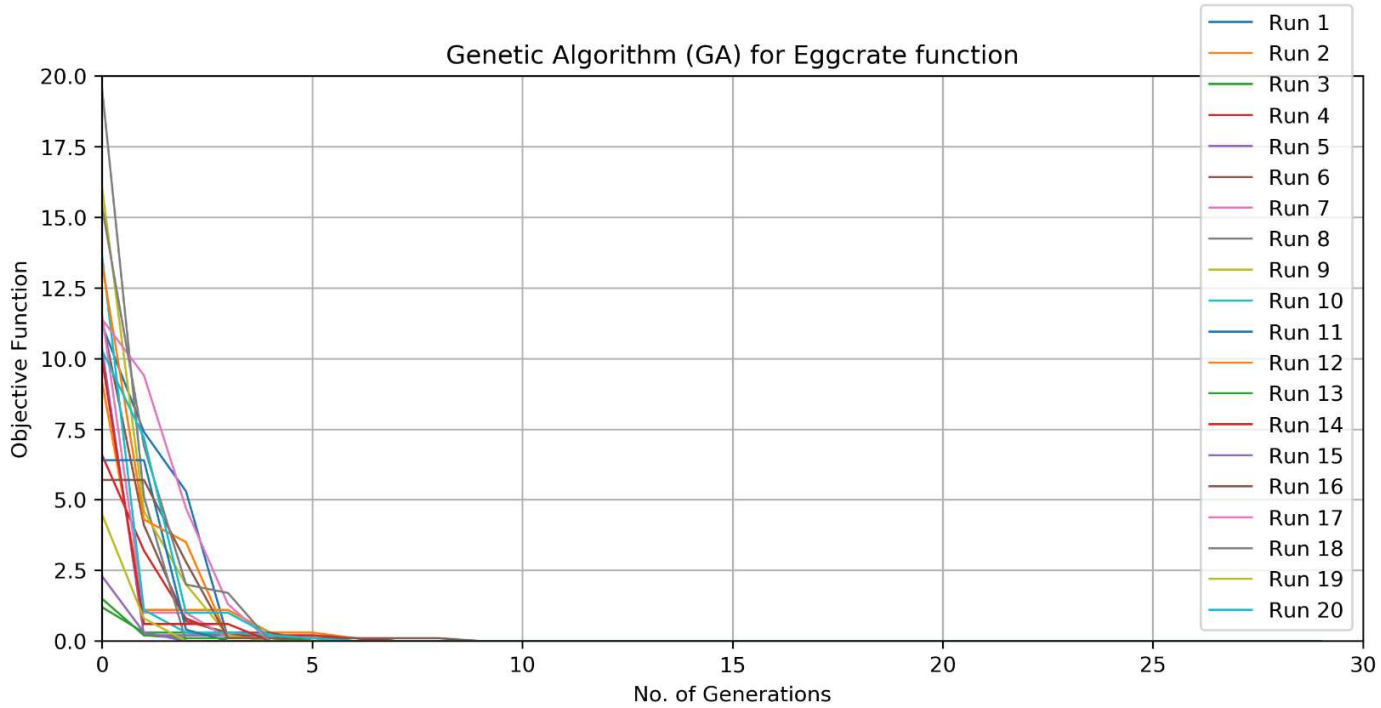
Run 0:, Best Cost 0.0:, Min X is 0.9:, Min Y is 0.9
Run 1:, Best Cost 0.0:, Min X is 1.0:, Min Y is 0.9
Run 2:, Best Cost 0.0:, Min X is 1.0:, Min Y is 0.9
Run 3:, Best Cost 0.0:, Min X is 1.0:, Min Y is 1.0
Run 4:, Best Cost 0.0:, Min X is 1.0:, Min Y is 1.1
Run 5:, Best Cost 0.1:, Min X is 0.7:, Min Y is 0.5
Run 6:, Best Cost 0.0:, Min X is 1.0:, Min Y is 1.1
Run 7:, Best Cost 0.0:, Min X is 1.0:, Min Y is 1.0
Run 8:, Best Cost 0.0:, Min X is 0.9:, Min Y is 0.9
Run 9:, Best Cost 0.0:, Min X is 0.9:, Min Y is 0.8

Run 10:, Best Cost 0.0:, Min X is 0.9:, Min Y is 0.8
Run 11:, Best Cost 0.0:, Min X is 1.0:, Min Y is 1.0
Run 12:, Best Cost 0.0:, Min X is 1.0:, Min Y is 1.0
Run 13:, Best Cost 0.0:, Min X is 1.0:, Min Y is 0.9
Run 14:, Best Cost 0.0:, Min X is 1.0:, Min Y is 1.0
Run 15:, Best Cost 0.0:, Min X is 0.9:, Min Y is 0.8
Run 16:, Best Cost 0.0:, Min X is 1.0:, Min Y is 0.9
Run 17:, Best Cost 0.0:, Min X is 1.0:, Min Y is 1.0
Run 18:, Best Cost 0.0:, Min X is 1.0:, Min Y is 1.0
Run 19:, Best Cost 0.0:, Min X is 0.9:, Min Y is 0.9

Ex. 3 Part 2 – Minimisation of the Eggcrate function using GA

The following parameters were tuned in order to achieve the optimal results. There were 20 runs in total.

params.max_gen = 30 # Max no. of generations
params.pop_num = 28 # Population number
params.prpn_children = 1 # Proportion of children
params.gamma = 0.1 # Crossover parameter
params.mu = 0.53 # Mutation variable
params.sigma = 0.3 # Mutation variable
params.beta = 1 # Variable for roulette wheel selection



The graph shows that most of the runs got to the optimal solution at around generation 8 much faster than with the Rosenbrock GA function. The optimal objective function value of 0.0 was achieved every run and the global minimum [0,0] was found every run as well. None of the runs got stuck at a local minimum as was found frequently with the Newton-CG algorithm. Although the gradient-based algorithm converged in slightly fewer iterations, it converged to a local minimum 70% of the time whereas with the genetic algorithm it never converges to the local minimum. The results of the genetic algorithm runs for the Eggcrate function are summarised below.

The Python scripts used were the following: *eggcrate_app.py, eggcrate_ga.py*

Eggcrate GA Results Summary

Run 0:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 1:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 2:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 3:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 4:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 5:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 6:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 7:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 8:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 9:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 10:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 11:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 12:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 13:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 14:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 15:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 16:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 17:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 18:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0
Run 19:, Best Cost 0.0:, Min X is 0.0:, Min Y is 0.0

Ex. 3 Part 3 – Golinski Speed Reducer Optimisation using GA

The Golinski Speed Reducer problem is different to the other two minimisation problems as it has constraints. Therefore, we need to compensate for constraint violations using a penalty or barrier function. There are several approaches in GA to handle constrained optimisation problems with penalty functions such as Death Penalty, Static Penalties, Dynamic Penalties, Adaptive Penalties and many more than I've listed here.

The idea of a penalty function is to replace the constrained function with an unconstrained one of the from:

Minimise{f(x) + cP(x)}

Where c is a positive constant and P(x) is a penalty function. As the Golinski function is defined by a number of inequality constraints the following penalty function was used:

$$P(\mathbf{x}) = \tfrac{1}{2}\sum_{i=1}^{m}(\max\{0, g_i(\mathbf{x})\}^2$$ [2]

Which gives a quadratic augmented objective function denoted by:

$$\theta(c,\mathbf{x}) \equiv f(\mathbf{x}) + cP(\mathbf{x}).$$ [2]

This resulted in the following augmented Golinski function (implemented in golinski_app.py):

[2] http://mat.uab.cat/~alseda/MasterOpt/const_opt.pdf

```
retval = (0.7854*x[0]*x[1]**2*( 3.3333*x[2]**2 + 14.9334*x[2] - 43.0934) \
    -1.5079*x[0]*(x[5]**2 + x[6]**2) + 7.477*(x[5]**3 + x[6]**3) + 0.7854*(x[3]*x[5]**2 + x[4]*x[6]**2))\
        + c * ((max(0, g1)**2)\
            + (max(0, g2)**2)\
                + (max(0, g3)**2)\
                    + (max(0,g4)**2)\
                        + (max(0,g5)**2)\
                            + (max(0,g6)**2)\
                                + (max(0,g7)**2)\
                                    + (max(0,g8)**2)\
                                        + (max(0,g9)**2)\
                                            + (max(0,g10)**2)\
                                                + (max(0,g11)**2))
```

The constant c was set to different values until a more or less consistent result was achieved. Which in the end resulted in a very large c value.

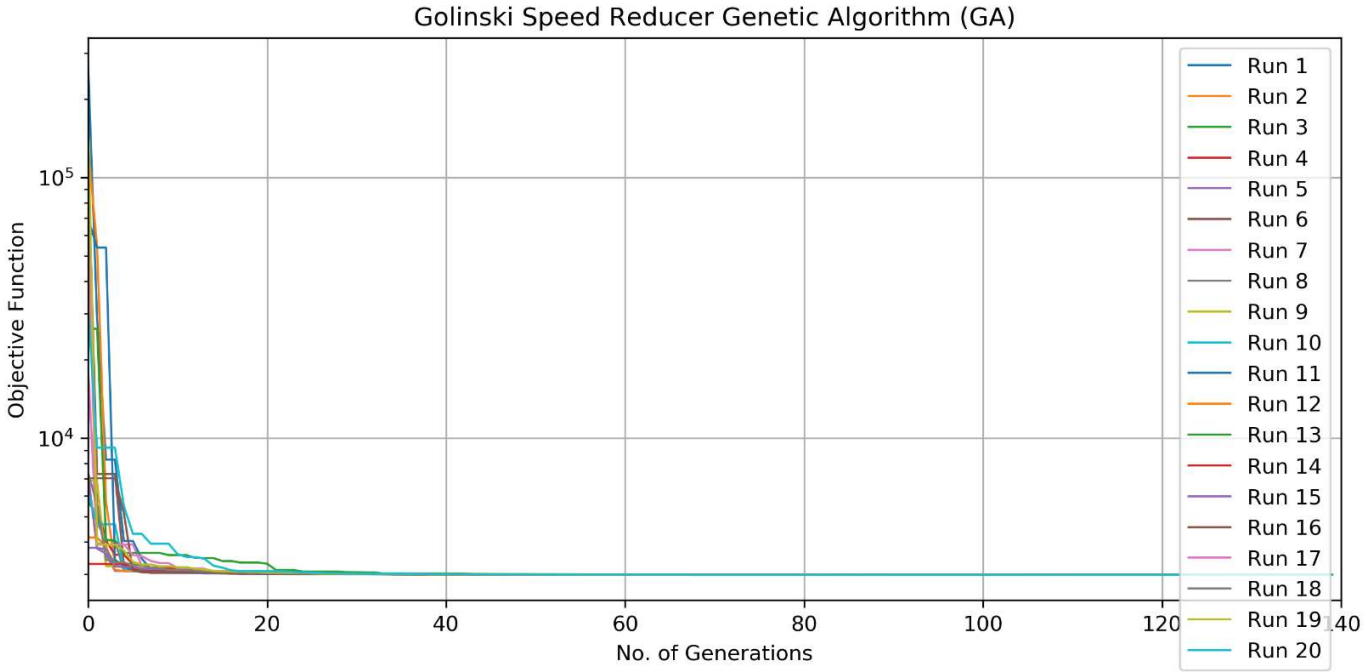
The following parameters were tuned in order to achieve the optimal results. There were 20 runs in total.

params.max_gen = 140 # Max no. of generations
params.pop_num = 50 # Population number
params.prpn_children = 1 # Proportion of children
params.gamma = 0.8 # Crossover parameter
params.mu = 0.1 # Mutation variable
params.sigma = 0.1 # Mutation variable
params.beta = 1 # Variable for roulette wheel selection

The penalty parameter used was the following:

c = 50000000 #penalty parameter

Below is a graph showing the generation history for each of the runs. Note that the graph below is on a logarithmic scale.



After tuning the other parameters, a very high value of c was required in order to achieve the global minimum for all variables and to arrive at an objective function optimum of 2994.34.

Although results close to the optimal objective function value were achieved with a population of 30-40, only a consistent value of 2994.34 (to 2 dcp.) was achieved with a population of 50. This value is very close to the 2994.35 found using the gradient-based algorithm. The optimal objective function and the global optimum [3.5, 0.7, 17.0, 7.3, 7.715, 3.35, 5.287] were found with every run.

The Python scripts used were the following: *golinski_app.py, golinski_ga.py*

Only the results of the final generation of each run is shown below.

Golinski GA Results Summary

Run no. 0, :Generation 139:, Best Cost 2994.338:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 1, :Generation 139:, Best Cost 2994.345:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 2, :Generation 139:, Best Cost 2994.34:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 3, :Generation 139:, Best Cost 2994.339:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 4, :Generation 139:, Best Cost 2994.341:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 5, :Generation 139:, Best Cost 2994.34:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 6, :Generation 139:, Best Cost 2994.343:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 7, :Generation 139:, Best Cost 2994.342:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 8, :Generation 139:, Best Cost 2994.338:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287

Run no. 9, :Generation 139:, Best Cost 2994.34:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 10, :Generation 139:, Best Cost 2994.339:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 11, :Generation 139:, Best Cost 2994.339:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 12, :Generation 139:, Best Cost 2994.343:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 13, :Generation 139:, Best Cost 2994.352:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 14, :Generation 139:, Best Cost 2994.343:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 15, :Generation 139:, Best Cost 2994.355:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.716:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 16, :Generation 139:, Best Cost 2994.338:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 17, :Generation 139:, Best Cost 2994.338:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 18, :Generation 139:, Best Cost 2994.34:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287
Run no. 19, :Generation 139:, Best Cost 2994.341:,
Min X1 is 3.5:, Min X2 is 0.7:, Min X3 is 17.0:, Min X4 is 7.3:, Min X5 is 7.715:, Min X6 is 3.35:, Min X7 is 5.287

It seemed strange to me that I needed such a large penalty constant in order to achieve the correct result so I decided to try out another genetic algorithm. This time I used Python PyPI's genetic algorithm package. However, I still required a penalty constant (c value) of 1000000 in order to reach the global minimum and the optimal objective function.

PyPI's genetic algorithm only displays a single best solution and objective function at the end and although the result was very close to the optimal objective function, the result was not consistent with consequent runs, I never got the same result twice even though it was close to the 2994 optimal value. However, the global minimum for the seven variables was always achieved.

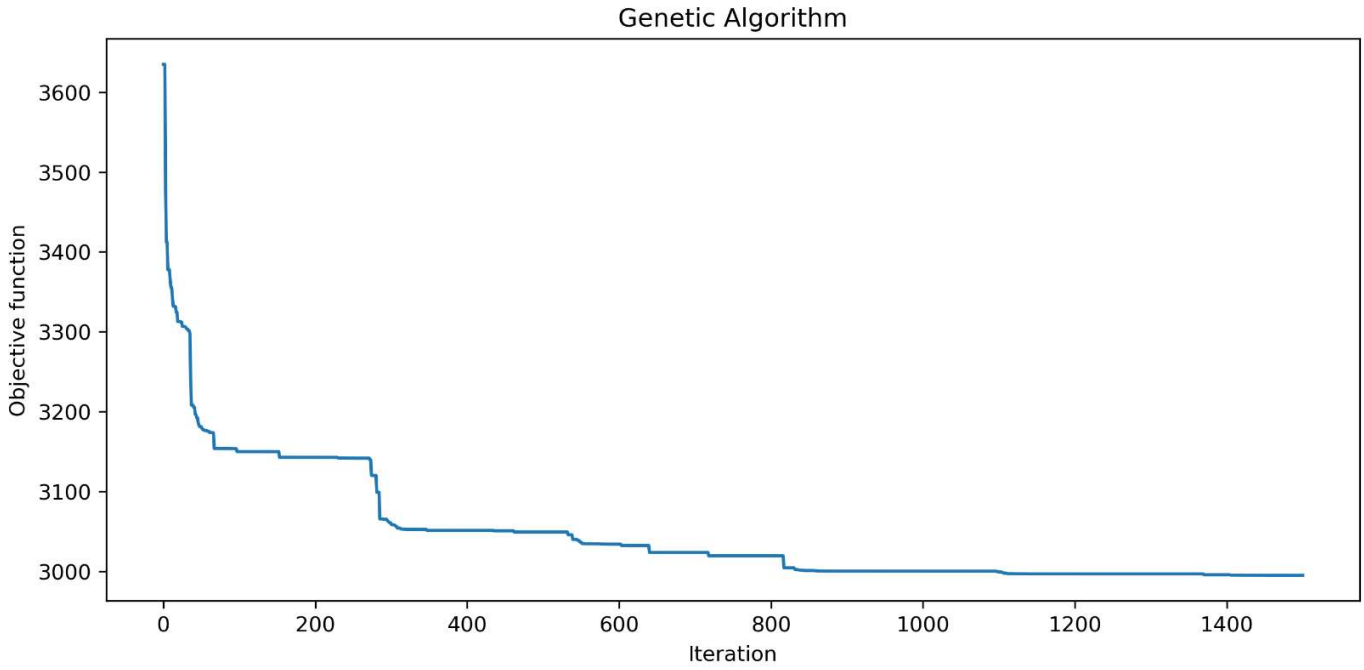
A much larger population and number of iterations (compared to the Yarpiz genetic algorithm) was required just to get close to the optimal values.

The Python script used was the following: *pypi_ga_golinski.py*

Below is the result of the tuned parameters for the PyPI genetic algorithm.

```
algorithm_param = {'max_num_iteration': 1500,\
        'population_size':500,\
        'mutation_probability':0.95,\
        'elit_ratio': 0.3,\
        'crossover_probability': 0.8,\
        'parents_portion': 0.3,\
        'crossover_type':'uniform',\
        'max_iteration_without_improv':None}
```

The below graph shows the iteration history of PyPI's genetic algorithm for the Golinski speed reducer problem.



As a result of the large population and number of iterations, PyPI's genetic algorithm was extremely slow.

I would say that the genetic algorithms worked well but required a lot tuning of the parameters to achieve the desired result. I expected the eggcrate function to be more heavy computationally because of all the local minima but it did not turn out to be as machine intensive as the Golinski problem which in my opinion required the most tuning and took the longest time to execute. However this may be due to the penalty function not being implemented correctly as the author is inexperienced with genetic algorithms and implementing penalty functions. Also, there was no elite parameter used with the Yarpiz genetic algorithm so this may also have influenced the Golinski results if implemented.

Comparison of the Gradient-Based and Genetic Algorithms

i. Dependence of answers on initial design vector (start point, initial population)

Algorithm Type	Rosenbrock	Eggcrate	Golinski Speed Reducer
Gradient Based	Not much dependence on the initial starting points. The global minimum and optimal objective values were always achieved no matter the initial starting points.	Difficult to say as the initial starting points close to the global minimum did reach the optimum but some other starting points not close to the global minimum also reached the optimum. The success rate was only 30%. Therefore I would say that there is a dependence on the initial starting points. It would probably be worthwhile carrying out a test where all the initial starting points are close to the actual global minimum in order to see if there is an improvement.	Not much dependence on the initial starting points, the global minimum was always found along with the optimum objective function.
Genetic	Not much dependence on the initial starting points. However the parameters required tuning to achieve success.	Not much dependence on the initial starting points. However the parameters required tuning to achieve success.	Again not very dependent on the initial starting points as the points did converge to the global minima. However a lot of time was spent tuning the parameters to achieve success.

ii. Computational effort (CPU time [sec] or FLOPS)

Algorithm Type	Rosenbrock	Eggcrate	Golinski Speed Reducer
Gradient Based	0.3585	0.0643	0.9251
Genetic	10.7326	5.0276	47.9105

iii. Convergence history

Algorithm Type	Rosenbrock	Eggcrate	Golinski Speed Reducer
Gradient Based	The initial starting points always converged to the global minimum and the objection function also reached the	The initial starting points always converged to either a local or global minimum in very few iterations and the objection function also reached	The initial starting points always converged to the global minima in very few iterations and the objection function also reached the optimal
Genetic	The initial starting points did not always converge to the global minimum but very close to it. However, the objective function did reach the optimal value but tuning of the parameters was required.	The initial starting points always converged to the global minimum and the objective function also reached the optimal value but tuning of the parameters was required.	Although the global minima and optimum were reached a significant amount of time was spent tuning the parameters. The execution time was also longer due to the increased population and number of iterations.

iv. Frequency at which the technique gets trapped in a local maximum

Algorithm Type	Rosenbrock	Eggcrate	Golinski Speed Reducer
Gradient Based	Never	70% of the time	Never
Genetic	Never	Never	Never

In conclusion, I would say that Gradient-Based optimisers are very efficient and require very few iterations to converge but they do have a tendency to get trapped at local minima as we saw with the Eggcrate function.

Genetic algorithms always converge to the global minimum but it may take a significant amount of time trying to tune the various parameters and also to execute them. Also, they seem to require more computational power as was seen with the Golinski problem. The PyPI genetic algorithm in particular used up a lot of machine power due to the large population and number of iterations. I have noticed that there are also differences between the performances of the genetic algorithms themselves. Big differences were noted between the Yarpiz genetic algorithm and the Python PyPI one. There are also bound to be differences with Matlab's or Octave implementation too.

Looking at the graphs produced from genetic algorithms it seems as if they converge close to the global minimum early on but they appear to take many more iterations to reach the actual global minimum consistently (as was seen with the Golinski implementation).

Both algorithms are good but I would say that the genetic algorithm is a more trial and error type of algorithm whereas gradient-based ones are more direct. Both require a good understanding of the search space.

All Python scripts are attached in the zip file. The reader must excuse the code quality as the author is a newbie to Python.