Exercise 1: Revenue Maximization Problem

We have been given that the airline wishes to develop a flight with 150 seats and p_1, p_2, p_3 are to be the three fare buckets which the company will have for its three class divisions. Also, D_1, D_2, D_3 represent the demand for seats or the maximum number of seats that each class division should have given the respective price bucket p_i. It is given to assume each Di to be a continuous variable for sake of simplicity and easy calculations. We will then round these off in the end after getting the final optimal prices to the nearest possible integer. The airline company is looking forward to maximizing its revenue which defines our cost function and can be represented as:

$$J(p_1, p_2, p_3) = p_1 D_1 + p_2 D_2 + p_3 D_3 \quad (1)$$

where $$D_i = a_i e^{-\frac{p_i}{a_i}} \quad for\ i = 1,2,3 \quad (2)$$

This function has to be maximized under the following equality constraint (i.e., assuming all the seats are occupied and which is quite obvious from the statement that company wants to maximize the revenue which is likely to be possible only if all the seats are occupied). But one can solve this as an inequality constraint as well, but to observe that this would be active since this is the bounding constraint on the objective function (e.g., to limit the revenue one needs to set a limit of the available capacity):

$$h = D_1 + D_2 + D_3 - 150$$

or

$$h(p) = \sum_{i=1}^{3} D_i - 150 = \sum_{i=1}^{3} a_i e^{-\frac{p_i}{a_i}} - 150 = 0 \quad (3)$$

This leads us to the optimization problem and the formal problem statement can be written as:

$$\begin{aligned} \min_{p} \quad & -J \\ s.t. \quad & h = 0 \\ & p_i \geq 0 \end{aligned}$$

(Note that we are using $-J$ because we want to maximize the cost/ objective function which is same as minimizing the opposite of the cost function. Hence, the notation.)

We can formulate the Lagrangian equation as

$$L(p, \lambda) = -J + \lambda h = -(p_1 D_1 + p_2 D_2 + p_3 D_3) + \lambda(D_1 + D_2 + D_3 - 150)$$

where λ is the lagrange multiplier.

Applying KKT optimality conditions for $\nabla L(p_i, \lambda) = 0$, we get:

$$p_i = a_i + \lambda, \ \forall i = 1,2,3 \quad (4)$$

The equations (1), (2) and (3) define the cost function, the demand for seats and the equality constraint, respectively. On solving these using *scipy.optimize.minimize* in python, we get:

$\boldsymbol{p_1 = \$156.75, p_2 = \$206.75,}$ and $\boldsymbol{p_3 = \$356.75}$

Substitute any of these in equation (4) to solve for $\boldsymbol{\lambda = 56.75}$

The maximum or optimal revenue thus is computed to be \$43670.8588 ~ **\$43670.86** with the demand or number of seats in each category as 20.85, 37.80, 91.34. On rounding these off to the nearest integer, we have $\boldsymbol{D_1 = 21, D_2 = 38,}$ and $\boldsymbol{D_3 = 91}$ totalling to 150.

Now, if we increase the total seats from 150 to 153, the equality constraint changes to

$$h = D_1 + D_2 + D_3 - 153 \qquad (5)$$

or if we consider that we can change N number of seats, we can come up with

$$h(p) = \sum_{i=1}^{3} D_i \; - \mathrm{N} = \sum_{i=1}^{3} \; a_i e^{-\frac{p_i}{a_i}} - N = 0 \qquad (6)$$

where N = number of seats

There are multiple approaches to solve this new problem to get the change.

1. Using the optimize function once again to get the new optimal prices and demand for seats.
2. Using linear approximation method.
3. Using sensitivity analysis

Approach 1:

Using the same minimize function from *SciPy*, we get the new values as D_1 = 21.68, D_2 = 38.79, and D_3 = 92.53, $\boldsymbol{p_1}$ **= \$152.87,** $\boldsymbol{p_2}$ **= \$202.87,** $\boldsymbol{p_3}$ **= \$352.87**, and maximum revenue = **\$ 43835.27**

After rounding-off the demand for seats, $\boldsymbol{D_1}$ **= 22,** $\boldsymbol{D_2}$ **= 39, and** $\boldsymbol{D_3}$ **= 92** totalling to 153.

We conclude that the company should make below amendments if it wants to add 3 more seats.
Change in fare buckets (in \$): p_1 = -\$3.88, p_2 = -\$3.88 and p_3 = -\$3.88
Change in seats: D_1 = 1, D_2 = 1 and D_3 = 1 (i.e. an increase of 1 seat per category)
To conclude, if the company decreases price bucket for each class by \$3.88 and increases one seat per class, it will increase its revenue by \$164.41.

Approach 2:

From (4) after differentiating w.r.t. λ, we get

$$\frac{\partial p_i}{\partial \lambda} = 1 \text{ or } \partial p_i = \partial \lambda \qquad (7)$$

i.e. *the rate of change of each price is same as the rate change of lagrange multiplier*.

Rearranging equation (6) and differentiating it w.r.t. p_i and using equation (7), we have

$$\frac{\partial \mathrm{N}}{\partial p_i} = \sum_{i=1}^{3} \; - e^{-\frac{p_i}{a_i}} = \frac{\partial \mathrm{N}}{\partial \lambda} \qquad (8)$$

After substituting the values for p_i and a_i (for i = 1,2,3), we get

$$\frac{\partial \mathrm{N}}{\partial \lambda} = \text{-0.765} \quad \text{or} \quad \frac{\partial \lambda}{\partial \mathrm{N}} = -1.3071$$

Now using linear approximation, $\Delta\lambda = \frac{\partial \lambda}{\partial \mathrm{N}} \Delta \mathrm{N} = \; -1.3071 \text{ x } 3 \; = \; -3.92.$

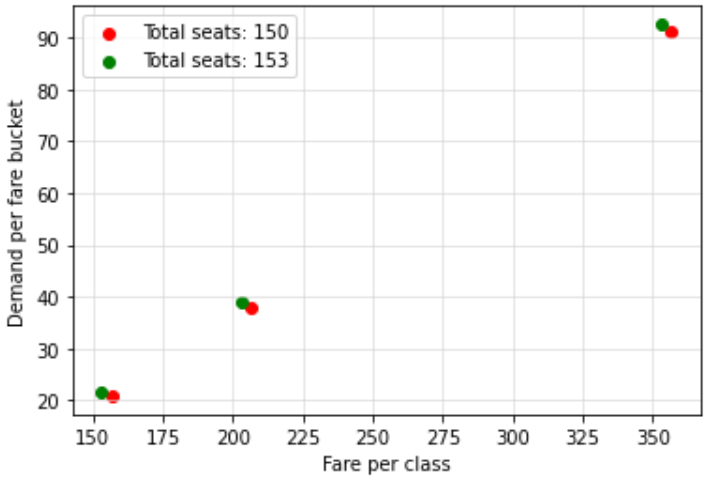
And thus, $\Delta p = \frac{\partial \mathrm{p_i}}{\partial \lambda} \Delta\lambda = [1,1,1] \text{ x } (-3.92) = [-3.92, -3.92, -3.92]$ i.e. the company should decrease each price bucket by \$3.92 which is almost same as the one obtained via approach 1.

Approach 3:

Sensitivity analysis measures the impact on the overall function and studies the behaviour of how other elements are affected when a small change is made to any of the element the function is made up of. From optimization theory, we know that the lagrange multiplier (λ) in case of maximization will be the positive of the sensitivity of the objective function to the constraint at the optimum i.e. in proportion to the same change as made to the constraint. Because our constraint (D1 + D2 + D3 - N) is dependent upon the variable element N (no. of seats), making any small change in N, say ∂n, will result in change in the constraint by the same amount and thus overall impact on the cost function will be in proportion as well. Thus, our cost function will change by an amount $-\lambda(\partial n)$.

i.e., $\partial(-J) = -\lambda(\partial n) = 56.75 \times 3 = 170.25$

The overall revenue will increase by $170.25 if the company adds on 3 more seats.

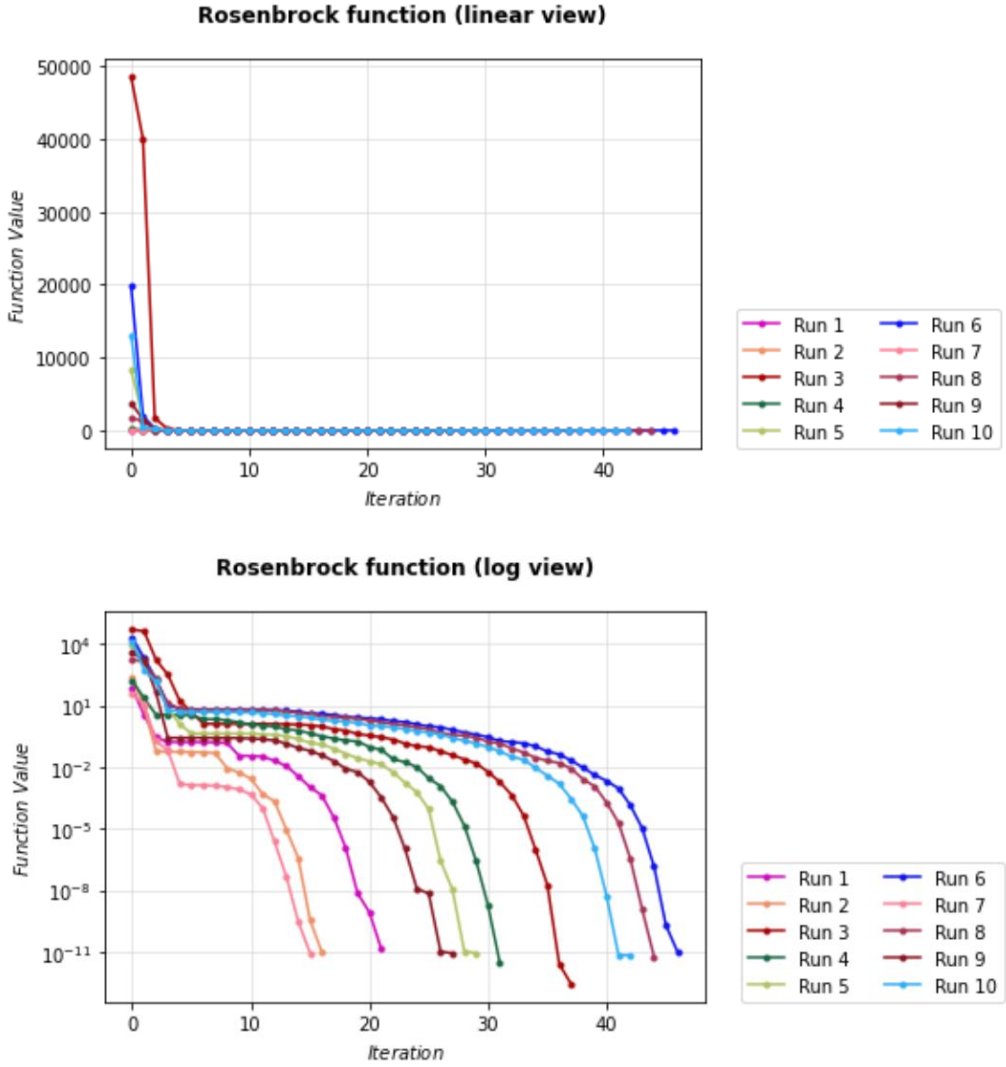


Exercise 2: Gradient-based methods to optimize problems

In minimizing **Rosenbrock function**, quasi-newton method (L-BFGS-B) implemented in python using *scipy.minimize.optimize* was used with random starting points chosen within bounds [-5, 5]. The program was run for 10 times and observations were recorded. In all cases, the algorithm converges successfully, and the global minimum was found within 6 significant digits.

Run#	Starting point (x0)	Starting Function value	Optimal point (x*)	Optimal function value	No. of iterations	Nature of optimal point	CPU Time (Sec)
1	1.638891, 1.863277	68.089488	0.999997, 0.999993	0.000000	21	Global	0.00457
2	-0.871564, -0.793824	244.822692	0.999997, 0.999994	0.000000	16	Global	0.00495
3	-4.735606, 0.404812	48526.015549	1.000000, 1.000001	0.000000	31	Global	0.01245
4	0.837555, -0.553005	157.404115	1.000000, 0.999999	0.000000	31	Global	0.00884
5	-3.618996, 4.043668	8217.860777	0.999997, 0.999994	0.000000	29	Global	0.00957
6	4.249324, 4.002681	19762.266137	0.999997, 0.999994	0.000000	46	Global	0.01307

7	-0.501461, -0.307431	33.490747	0.999997, 0.999994	0.000000	15	Global	0.00347
8	2.639523, 2.860663	1688.953904	0.999998, 0.999995	0.000000	44	Global	0.01334
9	-3.069559, 3.323824	3735.574138	0.999997, 0.999994	0.000000	27	Global	0.00452
10	3.440019, 0.398656	13082.043048	0.999997, 0.999995	0.000000	42	Global	0.00974

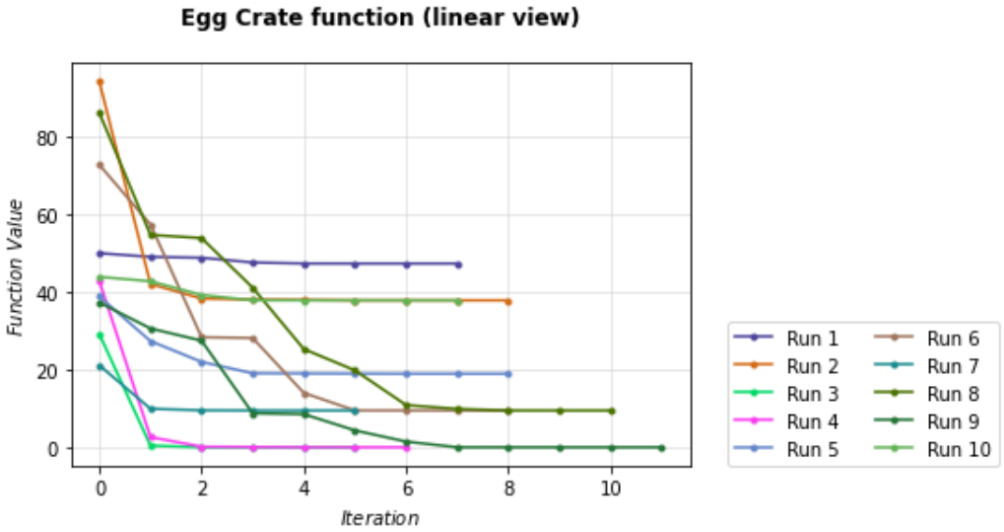


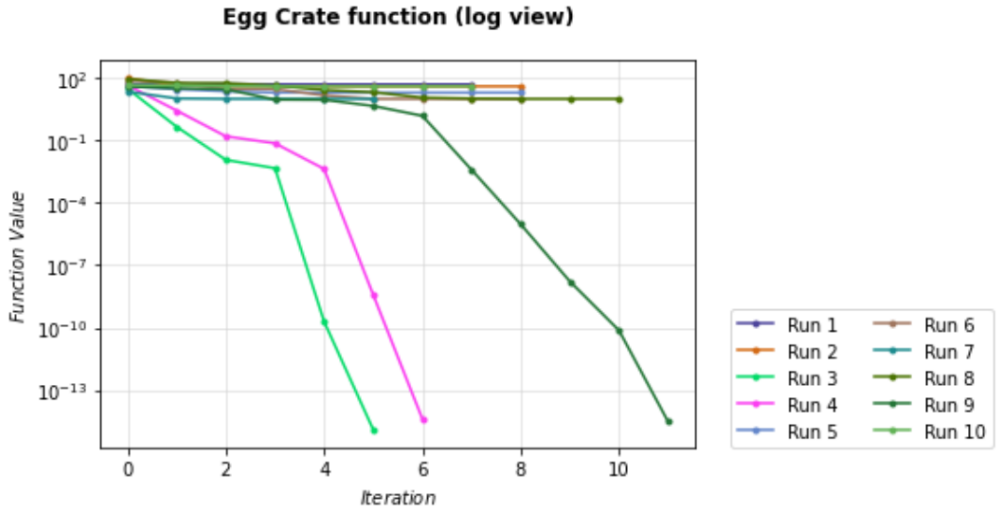
For **Egg Crate function**, the same algorithm was used, and it was noticed that gradient- based optimizer were stuck in the local optima, depending on the starting point that were chosen randomly within bounds [-2π, 2π]. Out of 10 runs, 7 were stuck in nearest local minimum while 3 found the global optimum.

It was concluded that gradient based methods get stuck in local minima at times and do not reach global optimum values.

Run#	**Starting point (x0)**	**Starting Function value**	**Optimal point (x*)**	**Optimal Function value**	**No. of iterations**	**Nature of optimal point**	**CPU Time (Sec)**
1	-5.684709, 3.137523	50.095432	-6.031424, -3.019602	47.417669	7	Local	0.0037

2	-4.986825, 4.639877	94.429580	-6.031424, 0.000000	37.929472	8	Local	0.00404
3	-3.577060, 3.332334	29.246990	-0.000000, -0.000000	0.000000	5	Global	0.001522
4	-3.313289, -4.028288	42.949895	0.000000, 0.000000	0.000000	6	Global	0.002759
5	1.732701, -2.723321	38.893694	3.019602, -3.019602	18.976395	8	Local	0.00231
6	1.992283, -4.907904	72.928749	3.019602, -0.000000	9.488197	8	Local	0.00354
7	-2.659793, 0.615779	21.161993	-3.019602, -0.000000	9.488197	5	Local	0.00204
8	5.834345, -4.757750	86.331546	-0.000000, 3.019602	9.488197	10	Local	0.00381
9	-3.921256, -3.091769	37.353924	0.000000, 0.000000	0.000000	11	Global	0.00465
10	-5.467543, 0.178303	43.967880	-6.031424, -0.000000	37.929472	7	Local	0.00379

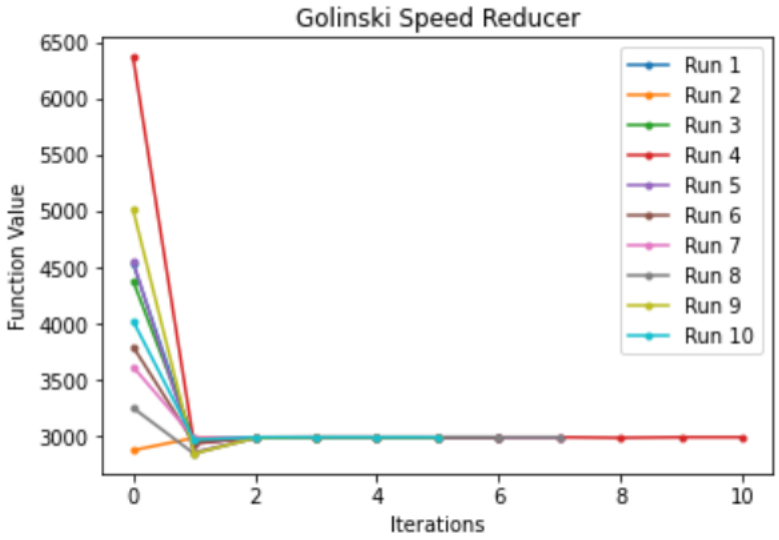




Finally, the **Golinski Speed Reducer** problem was solved using sequential least square quadratic programming approach (SLSQP) implemented using python.optimize.minimize and method=' SLSQP' that handles gradient-based constrained optimization problems. This problem has 11 inequality constraints in addition to 14 bound constraints and the objective is to minimize the weight of the speed reducer. The 10 starting points were picked at random as before and the algorithm converged to the optimum in all cases. The average optimum function value came out to be 2994.47.

Run#	Starting point (x0)	Starting function value	Optimal point (x*)	Optimal objective function	No. of iterations	Time (Sec)
1	[2.914301, 0.725399, 26.000000, 7.693571, 7.307665, 3.524833, 5.151876]	4535.819825	[3.499998, 0.700000, 17.000000, 7.300000, 7.715320, 3.350213, 5.286651]	2994.467622	6	0.01538
2	[3.289184, 0.701619, 17.000000, 8.080244, 7.403645, 3.264921, 5.257072]	2878.161131	[3.499956, 0.700000, 17.000000, 7.300000, 7.715312, 3.350205, 5.286655]	2994.451680	3	0.01205
3	[3.240819, 0.761802, 23.000000, 8.080795, 7.865247, 3.053914, 5.233260]	4382.075653	[3.499994, 0.700000, 17.000131, 7.300017, 7.715323, 3.350207, 5.286655]	2994.489618	4	0.01332
4	[3.442447, 0.779349, 27.000000, 7.896659, 8.094388, 3.687087, 5.666185]	6368.148517	[3.499999, 0.700000, 17.000000, 7.300000, 7.715320, 3.350214, 5.286653]	2994.470061	10	0.02676
5	[3.427577, 0.773341, 20.000000, 7.722538, 8.160341, 3.898382, 5.847531]	4552.291775	[3.499998, 0.700000, 17.000000, 7.300000, 7.715320, 3.350213, 5.286650]	2994.467113	7	0.02039
6	[3.057977, 0.718268, 23.000000, 7.534439, 7.813302, 3.048028, 5.074622]	3797.418940	[3.499997, 0.700000, 17.000000, 7.300000, 7.715320, 3.350213, 5.286649]	2994.465868	6	0.01401
7	[3.461898, 0.769745,	3619.291848	[3.500000, 0.700000,	2994.471389	4	0.01012

	18.000000, 8.235491, 7.886746, 3.482104. 5.368367]		17.000000, 7.300000, 7.715320, 3.350215, 5.286654]			
8	[2.781087, 0.731894, 18.000000, 8.179007, 7.581764, 3.020233, 5.776176]	3253.907765	[3.500000, 0.700000, 17.000000, 7.300000, 7.715320, 3.350215, 5.286654]	2994.471140	7	0.01957
9	[2.646268, 0.742785, 27.000000, 8.122423, 8.033381, 3.543504, 5.689108]	5014.046465	[3.499996, 0.700000, 17.000000, 7.300000, 7.715320, 3.350212, 5.286648]	2994.465167	5	0.01653
10	[3.409172, 0.729877, 22.000000, 8.252503, 7.585123, 3.528540, 5.030280]	4026.029944	[3.499997, 0.700000, 17.000000, 7.300000, 7.715320, 3.350213, 5.286650]	2994.466859	5	0.01502



It is evident from the graph that few values went below 2994.47 but because some of the constraints were not met, it went up again to find the optimized value that satisfies all the constraints.

Exercise 3: Heuristics methods to optimize problems from Ex2

There exists a lot of heuristic approaches like Stimulated Annealing (SA), Particle Swarm Optimization (PSO), Genetics Algorithm (GA) optimization, Evolutionary Algorithm (EA) optimization, etc. The benefit of these heuristic approaches lies in the fact that gradient based algorithms might get stuck in local minima whereas heuristic approaches tend to avoid the local minima and reach the global minima successfully giving optimized results as output.

In below sections we will see the same problems from exercise 2 solved using **PSO** approach. Later we will compare the results from both the approaches.

The heuristic approach used to solve the exercise is Particle Swarm Optimization. The Python module used is 'pyswarm' with just one main method *pso* (detailed description can be found over internet). The working model of this algorithm is very simple. Once the method is called, a starting point is chosen at random with each decision variable satisfying its bound constraints and with each function evaluation (happening internally in loop), a new best swarm position is calculated unless the algorithm finally converges to the optimized values. The implemented algorithm method internally performs many function evaluations, compares them, and keeps the new best (i.e. same locally optimized function value achieved after many swarm positions changes) out of them after each iteration. These results are what we see in the summary when *debug=True* option is enabled.

Also, it was observed that if the problem is unconstrained like Rosenbrock and Egg Crate functions, the convergence rate is very faster even with the default values (swarmsize = 100, omega = 0.5 = phip = phig) and completes in less than or equal to 25 iterations. On the other hand, if the problem is a constrained one with complexity like Golinksi Speed Reducer, the convergence depends upon a lot of combinatorial values for swarmsize, omega, phip, and phig. If the default values are not changed and correct values are not chosen properly, the algorithm either never converges even in 1 million iterations or the convergence happens too slow, say in 10k iterations. But, if good values are provided as input to the method, the algorithm converges very faster somewhere between 60-90 iterations of best/ new best values.

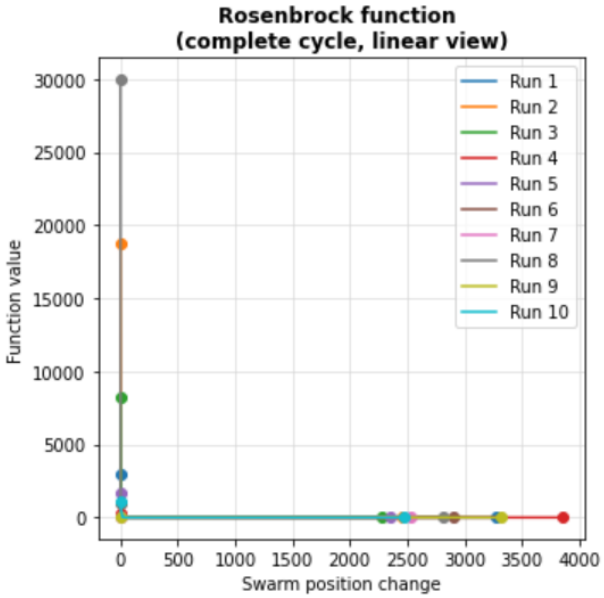
The following section describes in detail about the observations noticed and captured via this algorithm applied to Rosenbrock function, Egg Crate function, and Golinksi Speed Reducer.

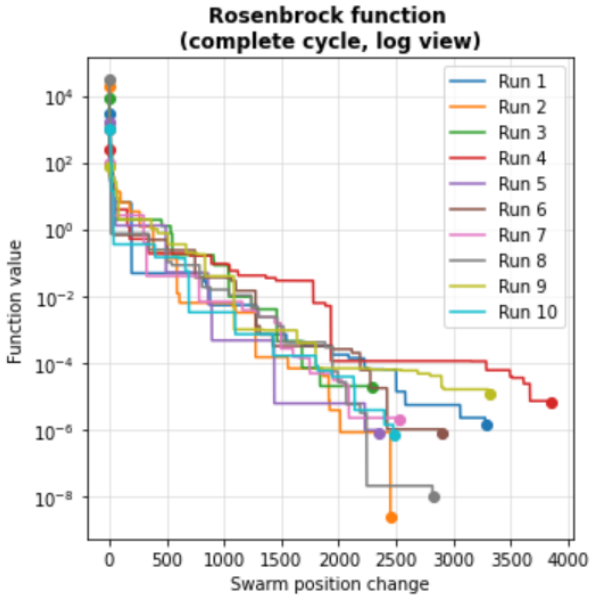
1. **Rosenbrock function:** $\boldsymbol{f(x_1, x_2) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1^2\right)}$

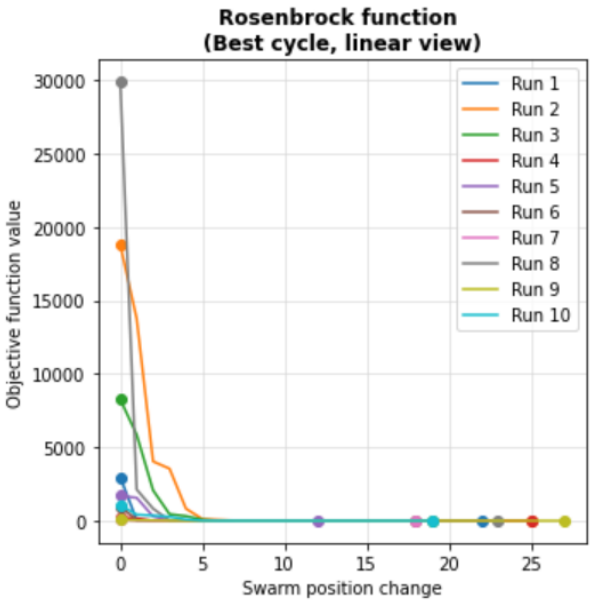
The default values were chosen for this function: swarmsize=100, omega=0.5, phig=0.5, phip=0.5 to reach at the best results optimized up to 8 decimal places. In all other combinations, either very higher iterations were reached before convergence, or the optimal decision variables and function values were only seen accurate only up to 2 decimal points (e.g. 0.99875432), but global optimum was reached. The ideal value range observed was phip=0.5-1.0, phig=0.5-1.0, omega=0.4-0.6, swarmsize > 100.
The algorithm was run for 10 times with different starting point at each run. With default tuned parameters, it was observed that algorithm converged to global optimum in all the runs.

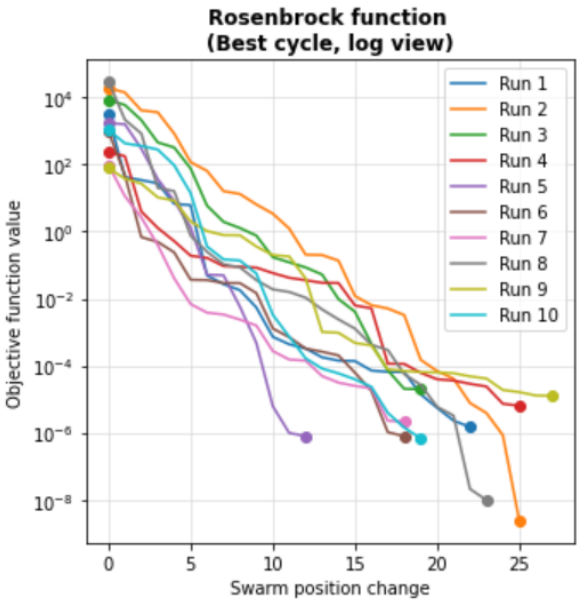
Run#	Starting point (x0)	Starting Function value	Optimal point (x*)	Optimal function value	No. of best/ new best values	Nature of optimal point	CPU Time (Sec)
1	-1.96234866 -1.59901726	2978.83969	1.00035220 1.00082177	0.000001	22	Global	0.14873
2	-3.38817658 -2.2239295	18798.31341	1.00003839 1.00007985	0.000000	25	Global	0.11265

3	-2.11680529 -4.60872486	8271.77821	1.00232746 1.00428017	0.000020	19	Global	0.10711
4	-2.17280983 3.22511139	233.865681	1.00244197 1.00495892	0.000006	25	Global	0.16896
5	2.91696859 4.3959019	1695.19030	1.00060343 1.00127352	0.000000	12	Global	0.09800
6	-0.12475987 3.15336826	985.845996	0.99988528 0.99985919	0.000000	17	Global	0.14617
7	2.39241051 4.7532762	96.0970744	0.99998720 1.00012133	0.000002	17	Global	0.11852
8	-4.53842627 3.30106389	29946.697581	0.99991511 0.99982493	0.000000	23	Global	0.12989
9	-1.57192157 1.63128189	77.116920	0.99649237 0.99305775	0.000012	27	Global	0.16196
10	-0.4599489 -3.04949654	1065.57585	0.99919179 0.99840409	0.000000	24	Global	0.11129



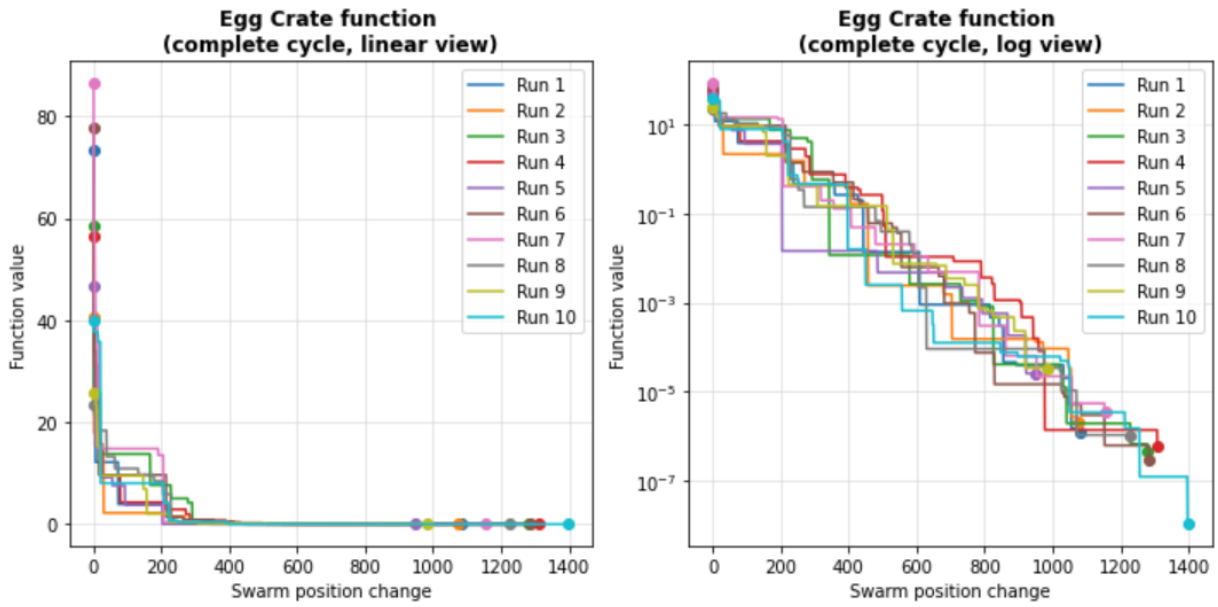


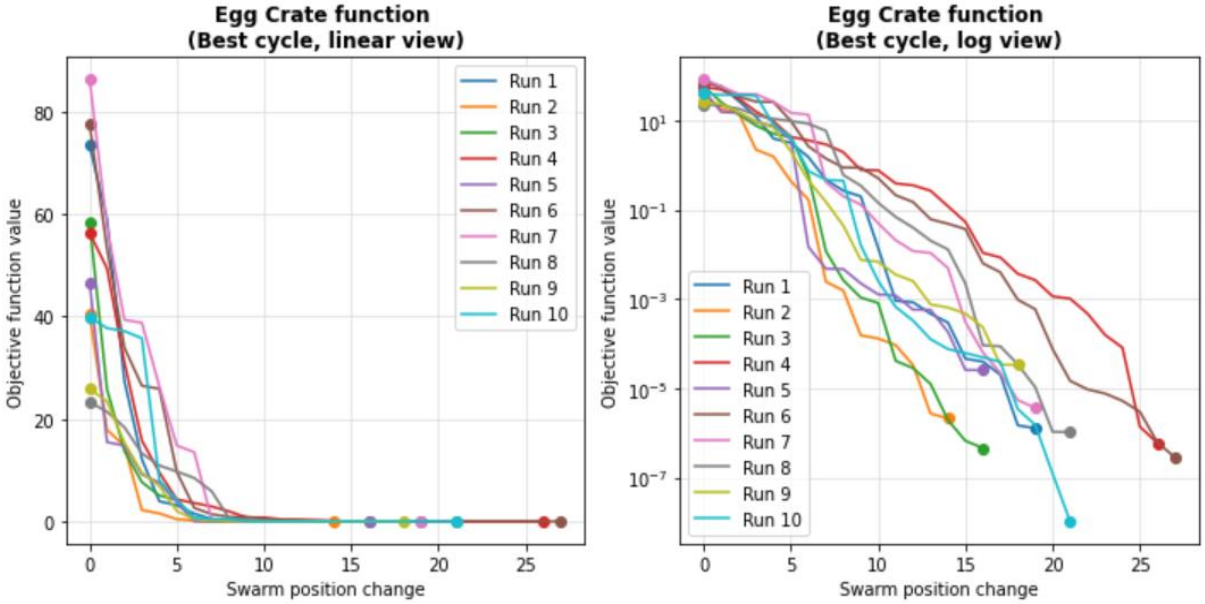




2. **Egg Crate function:** $f(x_1, x_2) = x_1^2 + x_2^2 + 25(sin^2 x_1 + sin^2 x_2)$

Run#	Starting point (x0)	Starting Function value	Optimal point (x*)	Optimal function value	No. of best positions since beginning	Nature of optimal point	CPU Time (Sec)
1	-4.948598 -3.846147	73.400056	-0.000215 -0.000043	0.000001	21	Global	0.08947
2	-5.695790 0.114829	40.461695	0.000177 -0.000223	0.000002	21	Global	0.07472
3	-3.648235 -5.781158	58.406889	-0.000125 -0.000044	0.000000	25	Global	0.09480
4	0.908872 5.707838	56.362885	0.000042 -0.000145	0.000000	26	Global	0.09790
5	3.1452776 4.2051885	46.678330	-0.000207 -0.000968	0.000002	29	Global	0.06170
6	5.525957 -4.155624	77.620279	0.000033 -0.000009	0.000000	46	Global	0.09948
7	5.262097 -5.659470	86.423721	0.000222 0.000299	0.000003	15	Global	0.08822
8	0.277298 3.721690	23.311946	-0.000003 -0.000197	0.000001	44	Global	0.09072
9	3.462641 3.289889	25.848501	-0.000809 -0.000787	0.000003	27	Global	0.06939
10	2.160735 -3.602048	39.842826	-0.000007 0.000002	0.000000	42	Global	0.10235





Golinksi Speed Reducer:

Like stated earlier, different behaviour could be observed with different combinations of the defined parameters (swarmsize, omega, phig, phip). Moreover, it was observed that ideal omega range came out to be 0.5-0.6. So, other parameters were changed and below observations were recorded to tune the parameters.

Higher swarmsize (>=500) and Lower phip, phig (0.7-0.8)	Fast convergence
Lower swarmsize (<100) and Higher phig and phip (>=1.55)	Fast convergence
Higher swarmsize (>=500) and Higher phip, phig (1.8-2.2)	Slow convergence
Lower swarmsize (10-100) and Lower phip, phig (<1.5)	No convergence
Lower swarmsize (100) and Medium phip, phig (1.5-1.75)	Fast but not enough
Medium swarmsize (150-300) and Medium phip, phig (1.5-1.75)	Fast convergence (best iterations range)

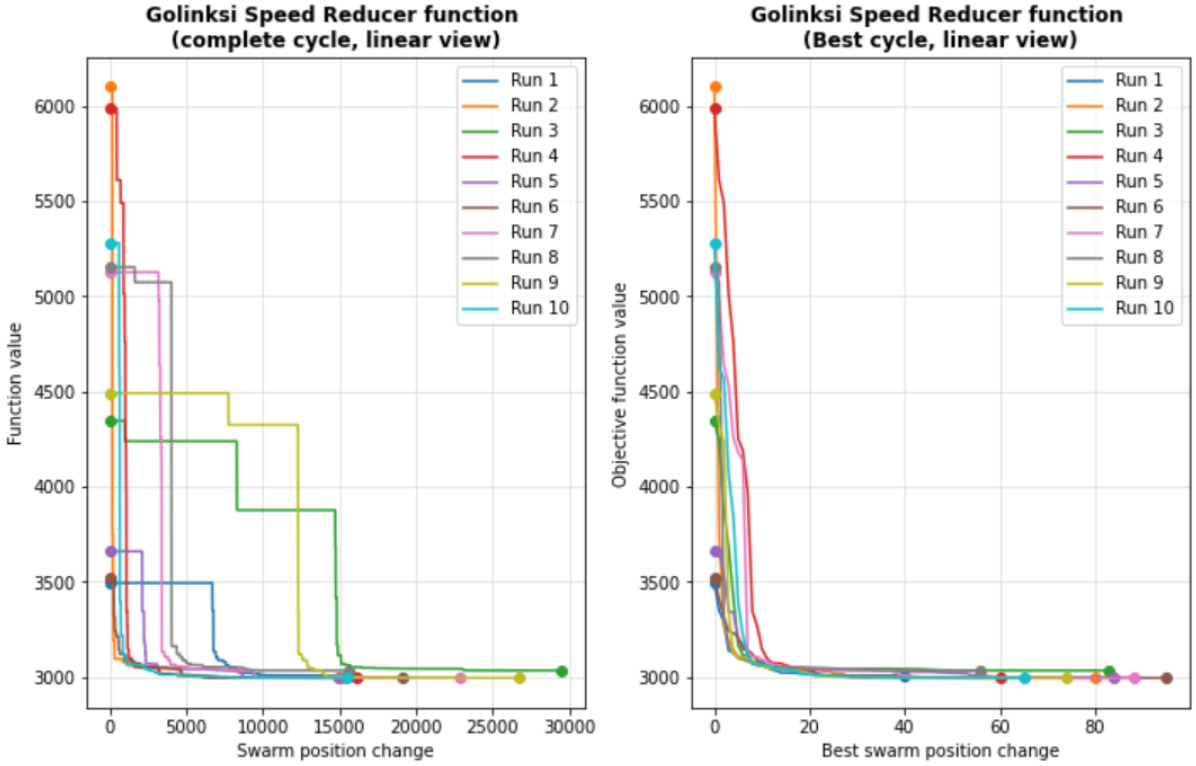
The ideal range for phip (particle confidence) and phig (swarm confidence) came out to be 1.5-2.2. For good convergence, the higher the swarmsize, better is the accuracy and convergence.

The results and graph below is based on the below tuned parameters.
swarmsize=200, omega=0.55, phig=1.52, phip=1.50, minstep=1e-5, minfunc=1e-05

Run#	**Starting point (x0)**	**Starting function value**	**Optimal point (x*)**	**Optimal function value**	**No. of best positions since beginning**	**Nature of optimal point**	**Time (Sec)**

1	[2.712976, 0.726899, 19.718452, 8.259598, 7.53648, 3.894512, 5.462583]	3493.646593	[3.500045, 0.7, 17, 7.3, 8.3 3.350235, 5.286861]	3007.461280	40	Global	1.02846
2	[3.427287, 0.756027, 2.901319, 8.195486, 8.059270, 3.410009, 5.414316]	6099.374739	[3.5, 0.7, 17, 7.3, 7.715420, 3.350217, 5.286659]	2994.479610	80	Global	1.01840
3	[2.828268, 0.777867, 23.352656, 7.362714, 8.262224, 3.501548, 5.262828]	4345.301880	[3.6, 0.7, 17, 7.3, 7.715344 3.350217, 5.286655]	3033.750083	83	Global	1.71698
4	[3.573455, 0.758840, 26.436600, 8.065170, 8.0474883, 3.818447, 5.426547]	5985.846695	[3.5, 0.7, 17, 7.3, 7.715330, 3.350217, 5.286657]	2994.473982	60	Global	0.96717
5	[2.606614, 0.750035, 20.522171, 7.814243, 8.151527, 3.713224, 5.511070]	3660.060219	[3.5, 0.7, 17, 7.3, 7.715443, 3.350224, 5.286673]	2994.494492	84	Global	0.90256
6	[2.970691 0.731202, 20.126181, 7.826279, 7.455571, 3.887803, 5.140098]	3521.942806	[3.5, 0.7, 17, 7.3, 7.715331, 3.350225, 5.286656]	2994.477404	97	Global	1.20500
7	[2.901078, 0.797520, 23.684643, 7.586304, 8.026023, 3.818828, 5.803342]	5125.967275	[3.5, 0.7, 17, 7.304735, 7.715328, 3.350225, 5.286655]	2994.515947	88	Global	1.39075
8	[2.654450, 0.725228, 27.937149, 7.570512, 8.211084, 3.553943, 5.785316]	5152.899280	[3.6, 0.7, 17, 7.3, 7.715402, 3.350222, 5.286663]	3033.757755	57	Global	0.91634

9	3.313922, 0.746417, 23.158171, 7.498814, 7.480947, 3.161113, 5.417705	4491.246882	[3.5, 0.7, 17, 7.3, 7.715330, 3.350216, 5.286657]	2994.473565	74	Global	1.57148
10	[3.541375, 0.756945, 25.239955, 7.342442, 7.686483, 3.435512, 5.176152]	5280.026102	[3.5, 0.7, 17, 7.3, 7.715410, 3.350217, 5.286662]	2994.482534	64	Global	0.94740



In all the cases the method converged to find the optimal minimum weight of around 2994.47 with design/ decision variables [3.5, 0.7, 17.0, 7.3, 7.7154, 3.3502, 5.2867]. We can see that this result is almost same as that of one achieved via gradient-based algorithm.

Comparison between results from Exercise 2 and Exercise 3

(i) Dependence on the initial design vector/ swarmsize:

Problem Name	Gradient-based Optimizer	PSO
Rosenbrock function	**Low**: Though farther design variables took more iterations, they finally converged to the global minimum.	**Low**: any swarm size > 50 works best with tuning parameters set correctly as explained above.

Egg Crate function	**High**: Converged to global minima if the initial design variables are chosen close to optimal values, otherwise the algorithm got stuck in local minima.	**Low**: any swarm size > 50 works best with tuning parameters set correctly as explained above.
Golinski Speed Reducer	**Low**: All the cases converged to give the optimal result.	**High**: the result changes as swarmsize is changed and tuning of parameters also play a vital role in selecting initial swarmsize.

(ii) Computational effort in terms of CPU time (average of all the results **in seconds**):

Problem Name	Gradient-based Optimizer	PSO
Rosenbrock	0.0084	0.1302
Egg Crate	0.0032	0.87
Golinski Speed Reducer	0.0163	1.07

(iii) Convergence history:

Problem Name	Gradient-based Optimizer	PSO
Rosenbrock	Always converged to global minimum.	Converged, but efficiency depends on the tuning parameters selected.
Egg Crate	Always converged, but either a local or a global minimum.	Converged, but efficiency depends on the tuning parameters selected. Also, the time taken to converge was more because of colonized local minima.
Golinski Speed Reducer	Always converged to global minimum.	Converged, but efficiency depends on the tuning parameters selected.

(iv) Frequency of getting trapped in a local minimum:

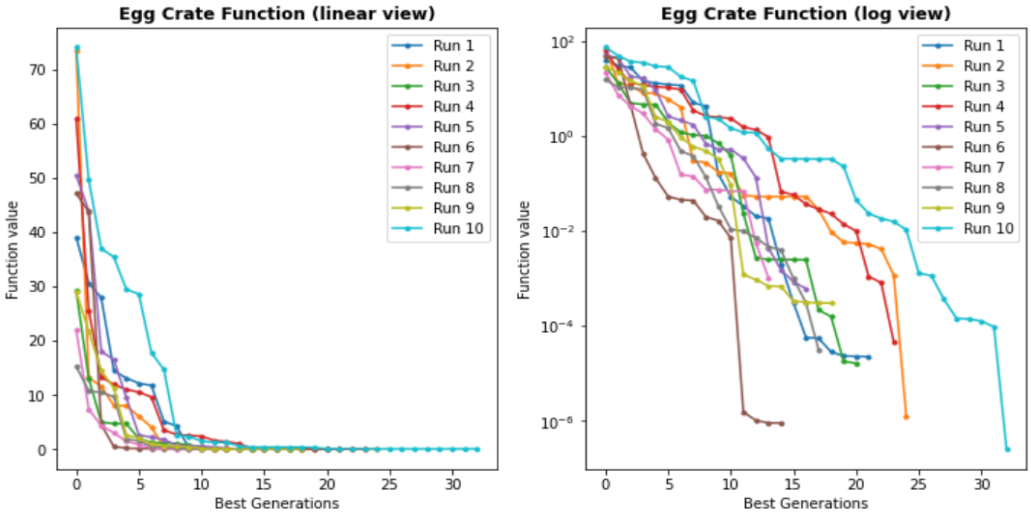
Problem Name	Gradient-based Optimizer	PSO
Rosenbrock	0	0
Egg Crate	50-80%	0
Golinski Speed Reducer	0	0

Let us have a look at an example using **Genetics Algorithms**. Though GA can be computed in various ways, the version used is Elitist Genetics Algorithm. The library used is *geneticalgorithm* implemented in PiPy (reference: https://pypi.org/project/geneticalgorithm/).

Egg Crate function converges to global minima with fewer best generations though it takes a lot of time in computing optimal values because of involvement of complex steps like initialization, evaluation, selection, crossover, mutation, parenting. The tuned parameters are 'max_num_iteration': None, 'population_size':100, 'mutation_probability':0.05, 'elit_ratio': 0.01, 'crossover_probability': 0.7, 'parents_portion': 0.3, 'crossover_type':'uniform', 'max_iteration_without_improv':200.

Run#	Starting point (x0)	Starting Function value	Optimal point (x*)	Optimal function value	No. of best positions since beginning	Nature of optimal point	CPU Time (Sec)

1	3.339287, -5.107265	39.051420	0.000919, -0.000415	0.000022	22	Global	5.66459
2	-4.973739, 4.951447	73.529773	-0.000131, -0.000631	0.000001	24	Global	3.85775
3	-1.981239, 1.806851	29.155000	0.000246, 0.002700	0.000016	20	Global	6.02472
4	-5.806008, -4.585873	60.997346	0.000032, 0.004773	0.000045	23	Global	3.83332
5	-3.920499, -4.664216	50.460613	0.004441, 0.006789	0.000605	16	Global	3.07244
6	-4.328514, 2.569037	47.122948	0.000008, 0.000665	0.000009	14	Global	5.55472
7	3.620672, -1.585864	21.936158	-0.006255, -0.002546	0.001030	13	Global	3.05604
8	-2.737841, 1.708666	15.255114	0.000821, 0.002605	0.000031	17	Global	6.11032
9	-1.859728, -1.265074	28.93880	-0.003402, -0.000048	0.000301	18	Global	6.57233
10	4.682586, 5.131099	74.067280	-0.000063, 0.000270	0.000000	33	Global	8.67973



We observe that the problem with GA is that it takes more computation time and thus results in more cost than PSO though the result is almost same. Also, with more complex problems involving higher dimensions like Golinski Speed Reducer which involves 7 decision variables and 11 inequality constraints, the time and cost is going to be more and tuning of parameters must be done properly and accurately. Also, if some constraints are not followed, we might need to add some penalty to make our algorithm work. This makes PSO and other available algorithms preferable over GA for such problems.

Summary:

Gradient Search Techniques

1. Efficient, repeatable, use gradient information
2. Can test solution via KKT (Optimality) conditions
3. Well suited for nonlinear problems with continuous variables
4. Can easily get trapped at local optima

(Meta-) Heuristic Techniques

1. Can be used for simple, complex, discrete, continuous and combinatorial variable problems
2. Use both a rule set (tuned parameters) and randomness
3. Do not use gradient information, instead search broadly
4. Avoid local optima, but are expensive