DATA SCIENCE TECH INSTITUTE

Metaheuristics Assignment

Andrea Sonnellini

S20 Cohort

This document details the reasoning I followed to solve the exercises of the assignment. It is made of 4 sections, the first 3 explain how I tackled the 3 exercises of the assignment, the last one recaps the pros and cons each algorithms showed in this work.
Along with this document I attach the following Jupyter notebooks which contain the code I run for my analysis:

- Final_Ex1_v3.ipynb
- Final_Ex2_rosen_eggcrate_gradient_v2.ipynb
- Final_Ex2_Golinski_gradient_method_v2.ipynb
- Final_Ex3_Rosenbrock.ipynb
- Final_Ex3_Eggcrate.ipynb
- Final_Ex3_Golinski.ipynb
- Final_Ex3_Golinski_parallel.ipynb

I will point out in this document which jupyter notebook is involved in the resolution of each exercise.

1 OPTIMAL PRICE FOR A FLIGHT WITH 150 PASSENGERS

1.1 Exercise 1 - reference for code: Final_Ex1_v3.ipynb

This problem is to use sensitivity analysis on revenue management for a very simplified airline pricing model. We will assume that an airline has one flight per day from Boston to Atlanta and they use an airplane that seats 150 people. The airline wishes to determine three prices, p_i ($i = 1,2,3$), one for seats in each of the three fare buckets it will use. The fare buckets are designed to maximize revenue by separating travelers into groups, for instance 14 days advance purchase, leisure travelers, and business travelers. The airline models demand for seats in each group using the formula: Where Di is the people that want to fly given price p_i, the remaining parameters are $a_1 = 100$, $a_2 = 150$, and $a_3 = 300$. Please note, for simplicity you may assume that each Di is a continuous variable.

- Formulate the revenue maximization problem for this flight as an optimization problem.

- What are the optimal prices and how many people are expected to buy a ticket in each fare bucket?
- Using sensitivity analysis, if the airline were to squeeze three additional seats onto this flight,
  - How much do you expect revenue to change?
  - By how much should the airline change each price?

1.2 Formulation of the optimization problem

The target is to maximize revenues from each flight, considering that each flight can carry 150 people max.
The number of people who will buy a ticket at price p_i is given by $D_i(p_i) = \exp^{-p_1/a_i}$ where $i = 1,2,3$.
The decision variables are p_1, p_2, p_3.

Revenues (to be maximized):

$$R(p_1,p_2,p_3) = \sum_{i=1}^{3} p_i D_i(p_i) = \sum_{i=1}^{3} p_i a_i e^{-p_i/a_i} \tag{1.1}$$

Constraints:

- $C_1(p_1) = p_1 \geq 0$
- $C_2(p_2) = p_2 \geq 0$
- $C_3(p_3) = p_3 \geq 0$
- $C_4(p_1,p_2,p_3) = \sum_{i=1}^{3} D_i(p_i) = \sum_{i=1}^{3} a_i e^{-p_i/a_i} \leq 150$

Cost function to be minimized is:

$$f(p_1,p_2,p_3) = -R(p_1,p_2,p_3) = -\sum_{i=1}^{3} p_i D_i(p_i) \tag{1.2}$$

The constraints can be re-written in the form $F(x) \leq 0$ as if we were applying the KT Theorem.

- $F_1(p_1) = -p_1 \leq 0$
- $F_2(p_2) = -p_2 \leq 0$
- $F_3(p_3) = -p_3 \leq 0$
- $F_4(p_1,p_2,p_3) = \sum_{i=1}^{3} D_i(p_i) - 150 \leq 0$

1.3 Qualitative study of the problem

Given that the cost function f is made of the decoupled sum of $-p_i D_i(p_i)$, I will plot the term $-p_1 D_1(p_1)$. From the plot in Fig. 1.1 we can see that each term $-p_i D_i(p_i)$ has a global minimum in a_i.

1.4 Resolution

To solve this optimization problem, I applied the KT-Theorem.

The first step was to check whether its hypothesis are satisfied:

- The Hessian Matrix of $f(p_1,p_2,p_3)$ is semidefinite positive in the region $p_i \leq 2a_i \ \forall i = 1,2,3$.

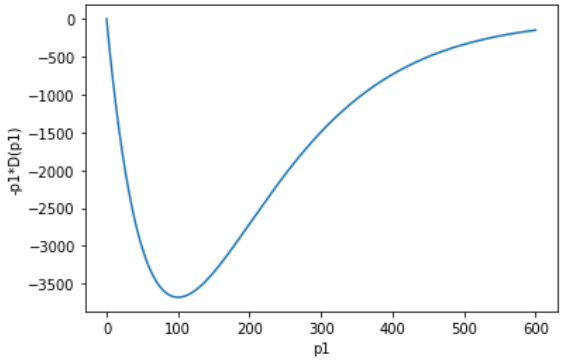

Figure 1.1: Plot of the term $-p_i D_i(p_i)$ from the Cost Function.

  - This means that f is convex in the region $p_i \leq 2a_i \forall i = 1,2,3$.

- $F_i(p_i)$ are always convex $\forall i = 1,2,3$.
- $F_4(p_1, p_2, p_3)$ is always convex.
- The point $(p_1, p_2, p_3) = (1,1,1)$ is such that all constraints functions are less or equal than 0

Given the hypothesis are all satisfied, I computed the KT relations and solved them numerically following the below criteria:

- I set to 0 all Lambda multipliers corresponding to the constraints F_1, F_2, F_3
- I solved numerically (using the function fsolve from the library scipy.optimize) the system of KT relations using various guess in input
  - As shown in figure 1.2 at every run the solutions are always the same

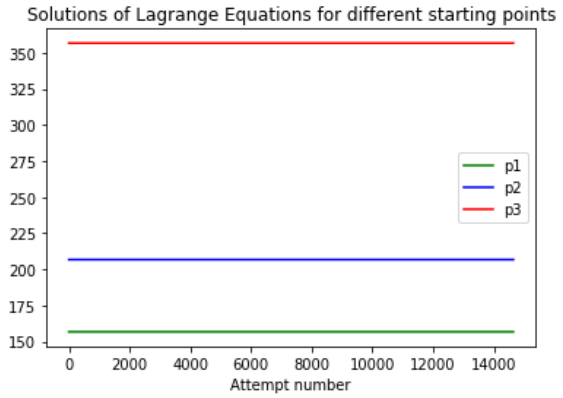

Figure 1.2: Solutions of Lagrange multiplier do not change changing the initial guess.

I report in the table below the results we obtained.

	Optimal Price	**Number of people buying a ticket**	**Revenues**
Backet a_1	156.74841942205822	20.856924256961985	3269.2899112843775
Backet a_2	206.74841952517284	37.80012831746722	7815.116787485078
Backet a_3	356.74841956052893	91.34294743994501	32586.452137200842
Total		150.00000001437422	43670.8588359703

1.5 Sensitivity Analysis

Using sensitivity analysis, if the airline were to squeeze three additional seats onto this flight:

- How much do you expect revenue to change?
- By how much should the airline change each price?

As stated on 29 May class (at 05:02:49 of the recording), sensitivity analysis means to check what is the variation of the cost function if one decision variable is changed while the others are fixed.

The table below shows how Revenues and people who will buy a ticket change when each price at a time decreases of 1%, and finally when all prices decrease of 1% simultaneously. For the last scenario we can see that the number of people who will buy a ticket increases approximately of 2.

	Absolute Variation of people	**Absolute Variation of Revenues**
Backet a_1 - 1%	+0.3295047116659866	+18.439950205462992
Backet a_2 - 1%	+0.5246149239913507	+29.2275054557349
Backet a_3 - 1 %	+1.0926991730613054	+60.055994642261794
All buckets -1% %	+1.946818808718632	107.72345030345969

Therefore for the case where there are 3 additional seats, prices should decrease of 3/2 x 1% = 1.5%. To check the validity of this statement, I compared the final adjusted prices (i.e. original prices decreased of 1.5%) with the ones obtained if I re-run the Lagrange multiplier system setting the constraint of having 153 people rather than 150.

	Sensitivity analysis	**Lagrange multiplier**
p_1	154.39719313072735	152.87443254
p_2	203.64719323229525	202.87443244
p_3	351.397193267121	352.87443251
Variation of revenues	+160.03511881642817	+164.41077931597738
Variation of people	+2.9297913779934674	+2.99999998567327

Figure 1.3: Comparison of prices obtained with sensitivity analysis VS Lagrange multiplier when consider flights with 153 people rather 150

The result of this comparison is shown in the above table. We can see that our predictions with Sensitivity analysis are quite close to what we found using Lagrange multiplier.

Another way to estimate how much prices should change if a flight can carry 153 people is to use Taylor expansion to the first order of the function $D(p_1, p_2, p_3) = \sum_{i=1}^{3} a_i e^{-p_i/a_i}$.
Let P_1, P_2, P_3 be the prices we found previously to maximize revenues when the airplane can carry 150 people.
When the airplane will carry 153 people, we assume that each price P_i will change of the same amount δP.
Using the Taylor expansion up to the first order of $D(P_1 + \delta P, P_2 + \delta P, P_3 + \delta P)$, we get:

$$\delta D = D(P_1 + \delta P, P_2 + \delta P, P_3 + \delta P) - D(P_1, P_2, P_3) = -\delta P \sum_{i=1}^{3} e^{-P_i/a_i}$$

It follows that:

$$\delta P = \frac{-\delta D}{\sum_{i=1}^{3} e^{-P_i/a_i}}$$

The minus sign shows that an increase in the number of people should lead to a decrease in prices.
Plugging in the formula $\delta D = 3$ and the values of the P_i we obtain $\delta P = -3.921329813411776$.
The sign minus means that prices should decrease.
Using the above variation, we get an increase in revenues equal to 166.372307641941, which is quite close to the value obtained using Lagrange multipliers.

2 Optimization with Gradient Descent algorithm

Numerically find the minimum (= optimal) feasible design vector x for each of the below three problems using a gradient search technique of your choice.
For each run record:

- the starting point you used
- the iteration history (objective value on y-axis and iteration number on x-axis)
- the final point at which the algorithm terminated
- whether or not the final solution is feasible.

Do at least 10 runs for each problem, but no more than 100. Discuss the results and insights you get from numerically solving these three nonlinear optimization problems.

2.1 The Rosenbrock (Banana) Function - reference for code: Final_Ex2_rosen_eggcrate_gradient_v2.ipynb

The Rosenbrock Function is:

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{2.1}$$

where x_1, x_2 will be considered for this problem to be in the interval [-5, 5].

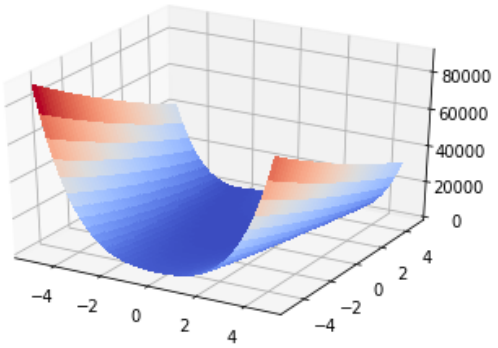

Figure 2.1: Shape of the Rosenbrock function.

This function has a known global minimum at [1, 1] with an optimal function value of zero. The target is to minimize it.

2.1.1 Resolution

Given that $f(x)$ is a polynom of degree 4, I choose the Newton method:

$$x_{k+1} = x_k - \epsilon(\nabla^2 f(x_1, x_2))^{-1} \nabla f(x_1, x_2) \tag{2.2}$$

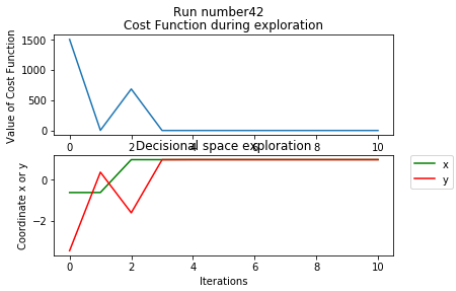

Figure 2.2: Explored points are always within the allowed decision space

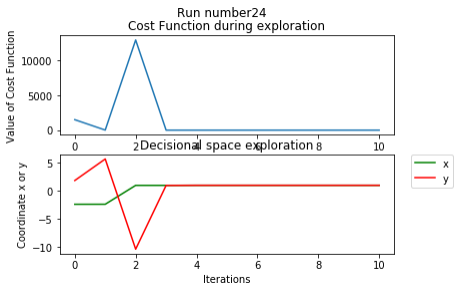

Figure 2.3: At iteration 2 the algorithm explores points outside the boundary [-5,5]x[-5,5]

I then computed the gradient and Hessian of the Rosenbrok function and coded my own Newton algorithm with a Stop Condition that is triggered when the Cost Function remains the same up to $5 \cdot 10^{-5}$ for 10 consecutive iterations.
Once the code was ready, I then applied the following procedure:

- I executed the Newton algorithm for 50 run
- Each run started from a point randomly chosen within the constrained decision space $[-5,5] \times [-5,5]$ and had a maximum number of iteration set to 10 (please see next section for an explanation about this value)
- ϵ was always set to 1.

2.1.2 Results

	Newton Algo
Number of runs	50
Success rate to known Min. (%)	100
Best value across all runs %	0.0
Best point of min	[1. 1.]
Starting point	[2.91032625, 3.53550349]
Average execution time (ms)	1.875

As shown in the above table, the algorithm always converged to the known point of minimum regardless the starting point .
Given the shape of the Rosenbrok function (quadratic), I quite expected such a good performance.
Looking at the the points in the decision space explored by the algorithm, I noticed that the algorithm:

- Converge in maximum 4 iterations, indeed Execution Time is very short
- Performs better in terms of intensification rather than diversification, i.e:
  - Either converges directly toward the point of minimum (see Fig. 2.2)
  - Or it explores one point outside the allowed region and right after it converges to the known minimum (see Fig. 2.3)

The above pattern allowed me not to need any projection mechanism over the allowed decision space or to have a large number if iteration per each run.

2.2 The Egg-Crate Function - reference for code: Final_Ex2_rosen_eggcrate_gradient_v2.ipynb

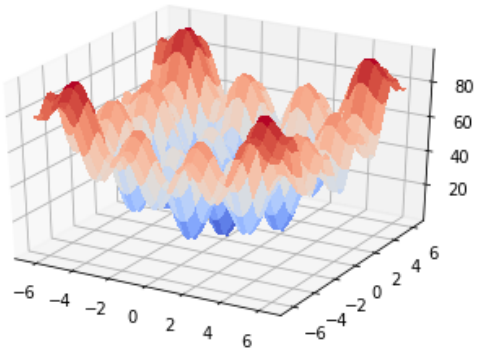

Figure 2.4: Shape of the Egg-Crate function.

The function to be minimized is:

$$f(x_1, x_2) = x_1^2 + x_2^2 + 25(\sin^2(x_1) + \sin^2(x_2))$$

where the two design variables x_1, x_2 are considered to be constrained within the lower and upper limits of $[-2\pi, 2\pi]$.
This function has a known minimum in $[0,0]$ and the minimum value is 0. As shown in Fig. 2.4 it also has several local minima.

2.2.1 Resolution

For this problem I re-used the Newton Algorithm that I implemented for the Banana (Rosenbrock) function, re-computing of course the gradient and the Hessian:

$$\nabla f(x_1, x_2) = (2x_1 + 50(\sin x_1 \cos x_1), 2x_2 + 50(\sin x_2 \cos x_2))$$

$$\nabla^2 f(x_1, x_2) = \begin{bmatrix} 2 + 50(2\cos^2 x_1 - 1) & 0 \\ 0 & 2 + 50(2\cos^2 x_2 - 1) \end{bmatrix}$$

I executed the Newton algorithm for 50 run, each run starting from a different random point within the constrainted region $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$. The maximum number of iterations was set to 100 per run (but most of the time the algorithm stopped before because it reached convergence, please see the next section).

2.2.2 Results

Grid of initial points	$[-2\pi, 2\pi] \times [-2\pi, 2\pi]$	$[-1, 1] \times [-1, 1]$
Number of runs	50	50
Success rate to known Min. (%)	0	42
Best value across all runs	9.488197339106259	7.147121655778748e-30
Best point of min	[3.34876179 x10^{-15}, -3.01960188]	[-5.19086968 x10^{-16}, 7.37428890 x10^{-17}]
Starting Point	[-0.65534075, -3.34891467]	[-0.58325062, 0.03352312]
Average execution time (ms)	5	5

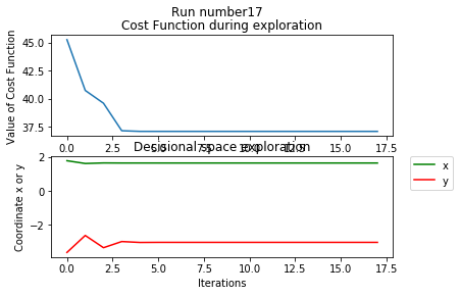

Figure 2.5: Convergence toward a local minimum

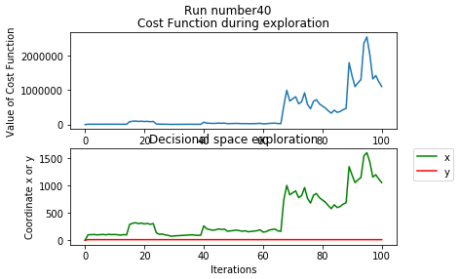

Figure 2.6: The algorithm does not converge in 100 iterations

As reported in the above table, the Algorithm is never able to reach the known minimum if the starting points are too far from it. Forcing the starting points to be close to the known minimum, the success rate increases from 0 to 42 %. Such poor performances were expected considering that the Egg Crater cost function has many local minima from which the Newton algorithm cannot escape.
In terms of Execution Time the algorithm is fast, but of course this does not counter-balance the poor performances reported above.
Focusing on the case where starting points are taken from $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$, we qualitatively observe that:

- Most of the time the algorithm converges to a point which we can assume to be a local minimum, e.g. see Fig. 2.5
- In approximately 4 cases the algorithm did not converge and started to explore portion of space much outside the allowed decision space, e.g. see Fig. 2.6

2.2.3 Possible improvements

In terms of improvements, we could have implemented a clip function that projects the points outside the boundaries over the surface of the decision space - this should have limited the possibility that the algorithm takes a path which has no chance at all to reconverge to the known minimum like the case shown in Fig 2.6.
Additionally, we could have tried to test various values of the step ϵ in the Newton algorithm at Eq. 2.2.

Nevertheless, given the large amount of local minima that characterizes the Egg Crater cost function, I think the algorithm is intrinsically not suitable for this type of cost functions and I do not expect the changes mentioned above to bring any concrete benefit.

2.3 The Golinski Function - reference for code: Final_Ex2_Golinski_gradient_method_v2.ipynb

The Golinski function provides in output the weight of a Gear Box that is described by 7 decision variables and 26 constraints:

- 14 constraints are the boundariy constraints (BC) for the 7 decision variables
- 12 constraints are non-linear coupled inequality constraints (NCIC)

All the details about this function and the constraints are reported in the text of the exercise.

2.3.1 Resolution

At first I tried to still use the same algorithm I coded for the previous 2 exercises, but soon I faced the difficulties related to the management of the constraints, in particular the 12 NCIC.
I managed the 14 BC "'simply"' implementing a clip function that projects any point outside the boundaries back to the allowed decision space.
Regarding the remaining 12 NCIC, I planned to add in the cost function penalization terms which are always zero except when a point violates the constraint.
For example if the original cost function is $f(x)$ and the function of one NCIC constraint is $F_1(x) \leq 0$, I defined a new cost function:

$$f(x) + \sigma \widetilde{F_1}(x) \tag{2.3}$$

where:

$$\widetilde{F_1}(x) = \begin{cases} 0 & if F_1 \leq 0 \\ F_1 & if F_1 > 0 \end{cases}$$

so that points outside the boundary are penalized. σ is a free parameter.
The drawback of this approach is that the computation of the Hessian matrix becomes extremely complicated. Because of that I attempted 2 alternative approaches:

- Run the Newton algorithm I coded for the previous problems without taking into account the 12 NCIC
- Code and run a standard Fixed-step Gradient Descent algorithm considering a penalized cost function like the one in Eq. 2.3 - in this way NCIC functions will be included without the need of the Hessian matrix

The first approach did not led to any useful insight, meaning that the algorithm always converged in maximum 2 iterations to one point on the border delimited by the 14 BC. Because of that, in the following I will discuss only the results obtained with the second approach, i.e. a standard Fixed-step Gradient Descent algorithm with a cost function which includes 12 NCIC via penalization terms.

The algorithm in object has 2 free parameters:

- The step ϵ of the algorithm
- The parameter σ coming from the cost function 2.3, which allows to increase or decrease the magnitude of the penalization term

I then run the algorithm on a grid of values for these 2 parameters considering the following range of values:

- $\epsilon \in [0.25, 0.5, 0.75, 1]$
- $\sigma \in [1, 50, 100, 200]$

For σ I chose this interval of values because I noticed the behavior of the algorithm changes when changing σ within this range. Regarding ϵ actually there was no specific driver, but I am aware we should try to estimate what is the scale of length at which the algorithm may be more sensitive to variations of ϵ.
For each couple (σ,ϵ) I launched 20 runs. Results are shown in the next section.

Number of runs	20
Value of σ	200
Value of ϵ	1
Success rate to known Min. (%)	0
Best value across all runs	1851.4154866393606 NOT FEASIBLE
Best point of min	[2.6 0.8 17. 8.3 7.3 2.9 5.] NOT FEASIBLE
Average execution time (ms)	12.5

Figure 2.7: The best point of min found for this set of run is not feasible because it violates NCIC F_5, F_6, F_8, F_{11}

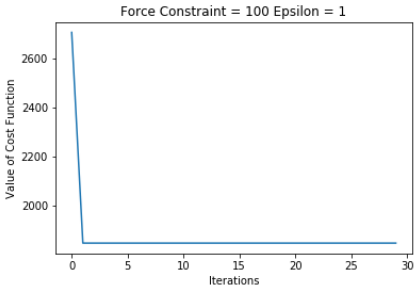

Figure 2.8: Force constraint (σ) = 100

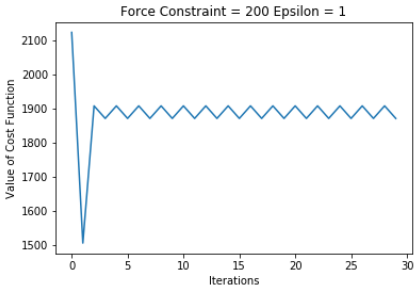

Figure 2.9: Force constraint (σ) = 200

2.3.2 Results

The above table shows results obtained for ($\epsilon = 1, \sigma = 200$), but other combinations show similar results. We can see that the algorithm never converges to a feasible point of the decision space, in particular it never converges to a point close to the known range of potential feasible min of the Golinski function.
Looking at the plot of the cost function at each run, we can identify 2 patterns:

- only σ seems to affect the behaviour of the algorithm
- for $\sigma \leq 100$, the algorithm immediately converges to a point which is within the 14 BC, but never within the 12 NCIC; I noticed that often these points are on the borders delimited by the 14 BC - e.g. see Fig. 2.8
- For $\sigma = 200$ the algorithm seems to bounce from one border to another and is not able to converge anymore, e.g. see Fig. 2.9; indeed, when the penalization term increases, points close to the boundaries delimited by the 12 NCIC becomes more and more penalized, forcing the algorithm to stay away from them

It really looks like the algorithm is not able to cope with the shape of the constrained decision space.

3 METAHEURISTICS ALGORITHMS

Repeat the numerical experiments from exercise 2, but this time using a heuristic technique of your choice (e.g. SA, GA ...). Explain how you "tuned" the heuristic algorithm. Both SA and GA. Compare your two algorithms (the gradient- search one and the heuristic one) from above quantitatively and qualitatively for the three problems as follows:

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

- Computational effort (CPU time [sec] or FLOPS)
- Convergence history
- Frequency at which the technique gets trapped in a local maximum

In order to answer this question, you do not need to implement your algorithms in some way BUT you MUST explain what algorithm is being used.
Describe not just your conclusions, but also the process you followed.
Do you think your conclusions would still apply for larger, more complex design optimization problems?

3.0.1 Strategy

For this exercise I decided to use the PSO algorithm.
I chose this algorithm because:

- I think its underlying working principle is more intuitive than other Metaheuristics Algo
- I was quite impressed when using it for the first time in class about the very good compromise between speed of execution and ability to find a good optimum point
- I understand that literature suggests already a range of values for its 3 tuning parameters (inertia factor, self confidence, swarm confidence) to tests, around 1; the numbers of swarms should be decided based on the problem.

Specifically I used the PSO algorithm implemented in Python package pyswarm because it allows to easily manage both equality and inequality constraints, a very important feature especially to optimize the the Golinski function.
The drawback of this package is that the algorithm does not return the values of the cost function at each iteration. Even enabling a higher debug level, it only prints on the screen the values of the cost function at each iteration, but this of course is not enough to draw a plot Cost Function VS iteration.

Due to the lack of time, I did not code my own PSO algorithm, but I intend to do it during the summer break to be more familiar with its pros and cons.

Regarding the approach I followed to solve this exercise, my first goal was to find the best set of values for the 4 parameters of the algorithm, i.e. the number of swarm and the value of the inertia, self-confidence and swarm-confidence parameters. To achieve this I:

- Defined a grid of values for this 4-plet of parameters
- Launched 20 runs for each combination
  - For Rosenbrock and Egg-Crate:
    - $N_s = [2, 4, 6, 8, 10]$
    - ω = [0.5, 0.75, 1]
    - ϕ = [0.5, 0.75, 1, 1.25, 1.5]
    - φ = [0.5, 0.75, 1, 1.25, 1.5]
  - For Golinski:
    - N_s = [10, 20, 30, 40, 50, 60]
    - ω = [0.5, 0.75, 1]
    - ϕ = [0.5, 0.75, 1, 1.2, 1.5]
    - φ = [0.5, 0.75, 1, 1.2, 1.5]
- Selected the 10 4-plet that led to a higher level of success rate, where success is defined as:

  - Hitting the known min for the Rosenbrock and Egg Crater function within a precision of 10^{-4}
  - For the Golinski function, find a point of min that lead to a value of the cost function close to the known points in literature, i.e. a value less than 3030; I chose this threshold based on the fact that a known possible solution is 2985, and other PSO algorithms found 3019;
  - If more than 10 set of 4-plet had the same success rate, I selected the 10 that led to lower values of the cost function
- Launch 50 runs for the selected 4-plet and select the best one still in terms of success rate
- Launch 100 runs for the top 4-plet and present result for this

After a brief review of the PSO main principles, I will present for each of the 3 problems considered in Exercise 2 the results obtained using a **tuned** PSO vs results from Gradient based algorithms.

3.0.2 Summary of PSO main principles

PSO is a P-method Metaheuristics, i.e. it uses a population of particles to explore the decision space modifying their position in parallel. In PSO the population of particles is called *swarm*. The main idea is to let the motion of each particle be affected by the best point found so far by the swarm and the best point found so far by the particle itself. From a quantitative point of view, the equation of motion of each particle is:

$$x^i_{k+1} = x^i_k + v^i_{k+1}\delta t \tag{3.1}$$

where x^i_{k+1} is the position of the $i-th$ particle at iteration $k+1$ which is obtained modifying x^i_k thanks to v^i_{k+1}, the particle velocity. The initial values of x^i_0 and v^i_0 are chosen randomly.
As mentioned before, v^i_{k+1} is updated considering at the same time:

- The current motion of the particle (i.e. its velocity at step k)
- The best position of the $i-th$ particle p^i so far
- The best position of the swarm at step k , namely p^g_k

v^i_{k+1} is indeed randomly updated via the following:

$$v^i_{k+1} = \omega v^i_k + \phi rand\frac{(p^i - x^i_k)}{\delta t} + \varphi rand\frac{(p^g_k - x^i_k)}{\delta t} \tag{3.2}$$

where the 3 parameters:

- ω is the inertia factor
- ϕ is the self-confidence
- φ is the swarm-confidence

are the parameters we can tune to modify the behaviour of the algorithm.
The documentation of pyswarm suggests to use for these parameters values in the range between 0 and 1.

3.1 The Rosenbrock (Banana) Function - PSO VS Newton - reference for code: Final_Ex3_Rosenbrock.ipynb

From table 3.1 we observe that the Rosenbrock function Newton is quicker and more precise than PSO in converging toward the minimum. I expected this because the Newton algorithm performs extremely well for quadratic cost functions with no local minima.

Algo	Newton	PSO
Parameters	step $\epsilon = 1$	$N_s = 10$ $\omega = 0.75$ $\phi = 1$ $\varphi = 1$
Number of runs	50	100
Success rate to known Min. (%)	100	100
Best min value of the cost F	0	1.283159 x 10^{-10}
Best point of min	[1,1]	[0.9999913694957971, 0.9999834727573873]
Dependence on starting point/initial design	Independent	Medium dependence on N_s
Approx Frequency trapped in local min. (%)	0	0
Average Execution time (ms)	1.875	60

Figure 3.1: Newton VS PSO for the optimization of the Rosenbrock Function; note the Frequency at which algo are trapped in local minima has been qualitatively estimated for the Newton looking at the plots Cost Function VS iteration

I think the same would hold even in dimension greater than 2 as far as the cost function is close to be a quadratic form with no local minima. I noticed PSO success rate depends on the number of particles meaning that with few particles the success rate drops at values lower than 50.

3.2 The Egg-Crate Function - PSO VS Newton - reference for code: Final_Ex3_Eggcrate.ipynb

Algo	Newton	PSO
Parameters	step $\epsilon = 1$	$N_s = 10$ $\omega = 0.75$ $\phi = 1.5$ $\varphi = 1.25$
Number of runs	50	100
Success rate to known Min. (%)	0	100
Best min value of the cost F	9.488197339106259	5.309186 x 10^{-13}
Best point of min	[3.3487e-15, -3.0196]	[5.530399230006079x 10^{-8} , -1.3176271412068505x 10^{-7}]
Dependence on starting point/initial design	Strong	Medium/Strong dependence on N_s
Approx Frequency trapped in local min. (%)	92	0
Average Execution time (ms)	5	42.96875

Figure 3.2: Newton VS PSO for the optimization of the Egg-Crate Function; note the Frequency at which algo are trapped in local minima has been qualitatively estimated for the Newton looking at the plots Cost Function VS iteration; in this table some decimals in the value of the cost function and the points of min have been removed to be able to plot the table within the page boundaries

The above table depicts very well how difficult is for a gradient-based algorithm to manage functions with multiple local minima. This type of algorithm is indeed not able to escape from local minima and can easily be trapped in them. Conversely, the PSO is always able to capture the known minimum regardless the presence of multiple local minima. Even with the Egg-Crater cost function, the success rate is quite dependent on N_s.

3.3 The Golinski Function - PSO VS Newton - reference for code: Final_Ex3_Golinski.ipynb Final_Ex3_Golinski_parallel.ipynb

The figures reported in table 3.3 show that the **tuned** PSO had much better performances than the Gradient-based algorithm. PSO was indeed able to find a feasible point of minimum in 80%

Algo	Fixed step Gradient-descent	PSO
Parameters	$\epsilon = 1$, $\sigma = 200$	$N_s = 60$ $\omega = 0.75$ $\phi = 1.5$ $\varphi = 1.5$
Number of runs	20	100
Success rate to known Min. (%)	0	80
Best min value of the cost F	1851.41548663 NOT FEASIBLE	2994.355027495922
Best point of min	NOT FEASIBLE	[3.5, 0.7, 17, 7.3, 7.71531993, 3.35021467, 5.28665447]
Dependence on starting point/initial design	Never converges	Strong dependence on all parameters
Approx Frequency trapped in local min.	Never converges	Cannot check
Average Execution time (ms)	12.5	1781.25

Figure 3.3: Newton VS PSO for the optimization of the Golinski Function; note the Frequency at which algo are trapped in local minima has been qualitatively estimated for the Newton looking at the plots Cost Function VS iteration.

of the runs, with a best value of 2994.355027495922 (the best known solution in literature that I am aware of is 2985), as opposed to the Gradient-based algorithm which was never able to converge to a feasible point.
On the other hand, tuning the 4 PSO parameters was challenging because many combinations of their values are possible, and the execution time of a run increases when the number of particles increases.
To tackle this problem, when launching the first set of runs on all possible combinations of the PSO parameters I:

- Limited the granularity of the grid of values to test
- Limited the number of runs for each combination - I re-increased this only upon the selection of the most performing values of the parameters
- Run 2 scripts in parallel, each of them testing different values of the grid

Another option could have been to test a limited number of combinations for these 4 parameters, choosing each of them randomly.
Another comment regarding the results obtained with PSO is that the best selected model had the nubmer of swarms, the Self confidence and the Swarm Confidence taking the maximum values we tested. This suggests that a next step could be to test higher values and check whether this improves performances.

4 Conclusions

The various scenarios studied in this assignment show that for simple and "'easily constrained"' quadratic-like cost functions, Gradient-based algorithms performs better than PSO (and I believe other metaheuristics algorithms as well) in terms of speed of execution, ease of implementation and accurateness of the results. Nevertheless as soon as the cost function or the decision space constraints are not as simple as the case of the Rosinski function, gradient based algorithms become completely useless because trapped in local minima or unable to manage and adapt to constraints. Conversely, the functioning principle of PSO, i.e. a set of particles that explores the decision space moving randomly but still having a collective memory of the best

collective and individual solutions found so far, let PSO overcome the limitation of gradient-based algorithms and achieve very good results **once tuned**.
Tuning PSO is challenging and time consuming given the high number of combination of values that its 4 parameters can take. Additionally, for large number of particles (60 in my analysis) PSO may need up to 2 seconds for 1 run.

In conclusion, as often mentioned during the class, each cost function/problem has its own peculiarities and the choice of the algorithm should be done considering such peculiarities VS the strengths/weaknesses of the available optimization algorithms. That being said, metaheuristics algorithms seem to be the best choice for complex non-linear constrained problems.