Meta-Heuristic Assignment S20

June 26, 2020

1 Exercice 1 - Airline pricing model

One flight per day From Boston to Atlanta / 150 people Three prices pi (i=1,2,3)

- category 1 : 14 days advance purchase
- category 2 : leisure travelers
- category 3 : business travelers

Demand : Di = ai*exp(-pi/ai), ai = (100, 150, 300), we estimate the Di functions are continuous

1.1 Revenue maximization problem

Here the three demand functions can be understood as the number of seats of each category that will be provided.

So the revenue is the product of the price of one seat multiplied by the number of seats sold, for each category. So

Revenue(p1,p2,p3) = p1 * D1(p1) + p2 * D2(p2) + p3 * D3(p3)

where

- pi is the price of the seat for the category i
- Di the demande for category i, depending on pi

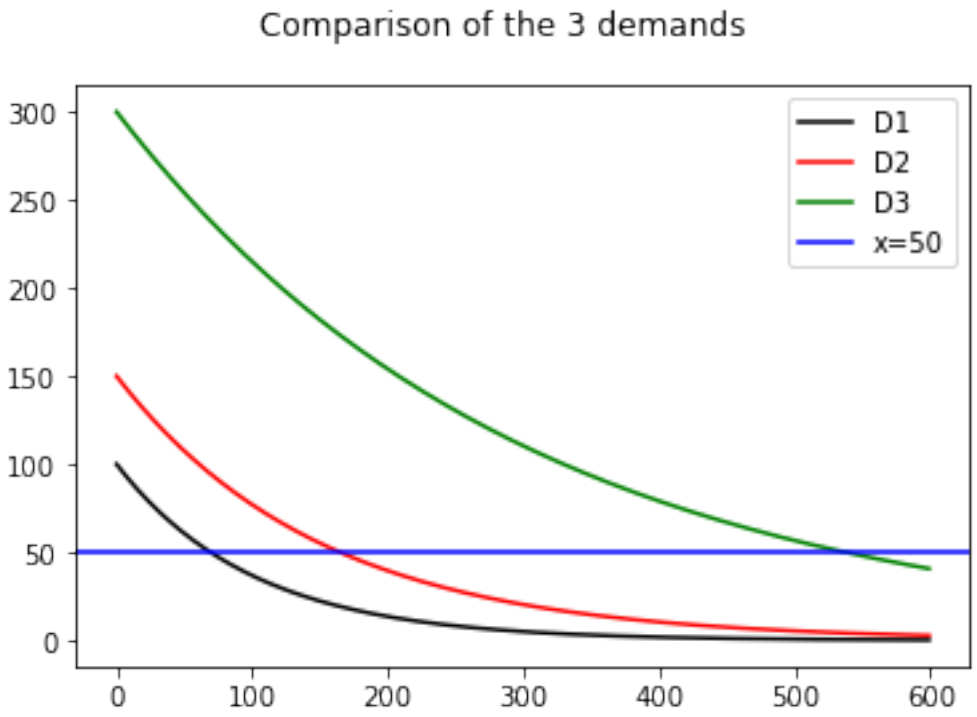
So the formulation of the problem is :

- Maximize the Revenue value (cost function) define just above
- we can assume that the universe search is for each pi (0, +infini(, even if in practice we will put a higher value to pi
- under the constraint : D1+D2+D3 = 150 : the number of seats are 150.

1.2 Optimal prices and number of people expected in each category

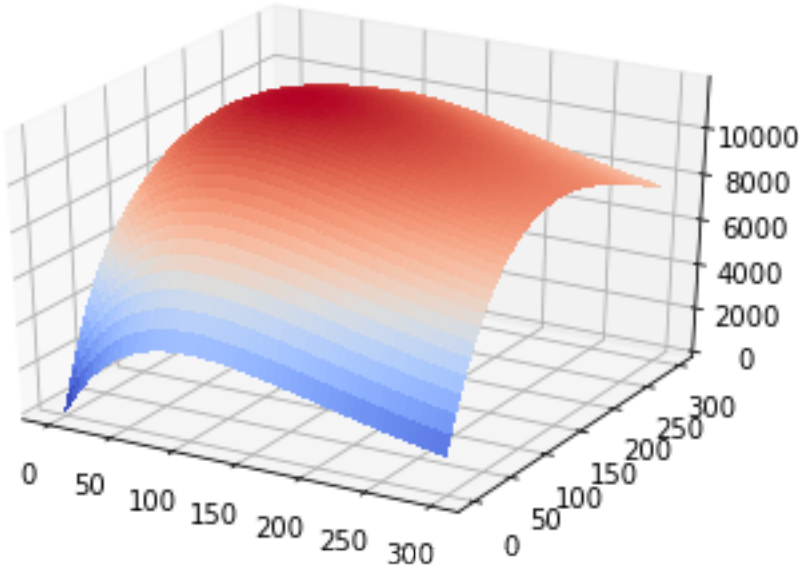
1.2.1 First let's have a look at demand function

- The three demans functions are decreasing from a to 0 when price vary from 0 to infinity
- The three curves crosses the horizontal line y=50 : That means that at least a "naive" solution exist to this problem where the number of allocated seats are the same for each category equal to 50.



1.2.2 Then let's have a look at the revenue function

As it is a 3 dimensionnal function, the visualisation in 4D is not possible, but we can have a look at the equivalent 2D function to this the revenue function in 3D Revenue2D = p1 * D1 + p2 * D2



1.2.3 First test with PSO algorithm

I copy/paste the definition of the pso function from

https://github.com/tisimst/pyswarm/blob/master/pyswarm/pso.py

And I slightly modify it to be able to remove and treat outup of the function regarding the reason why the iteration process was stopped. I also add new return values to be able to follow the convergence of the algorithm.

The PSO algorithm is adapted to continuous variable, quick and takes in account "natively" the ability to pass a constraint function. So I will use this algorithme. for the implementation :

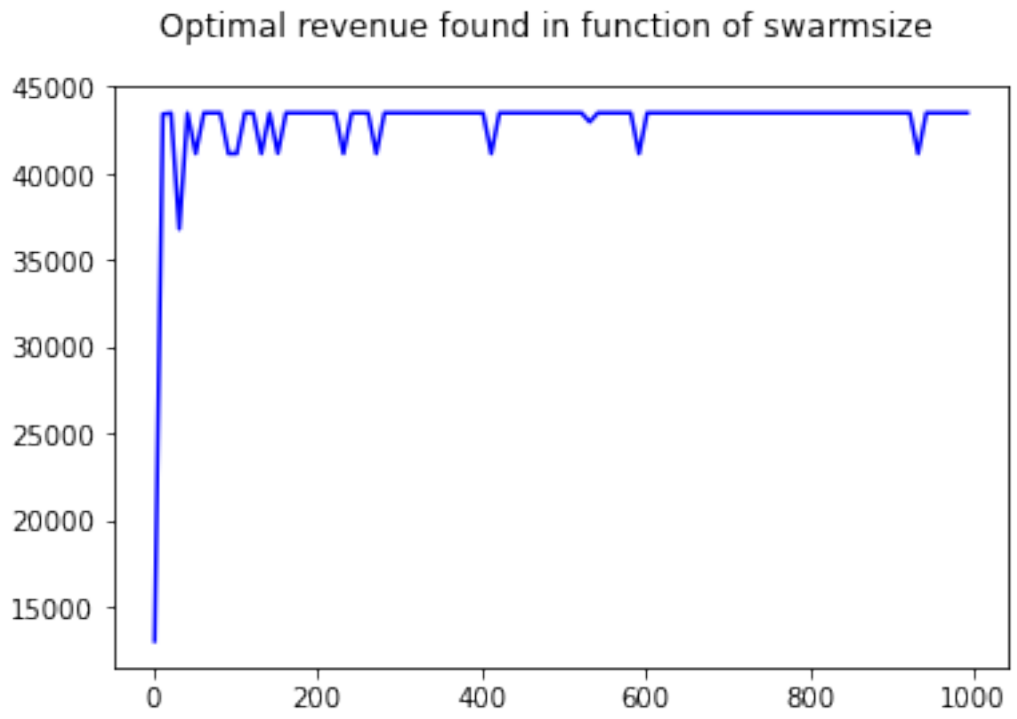
- the revenue function will be transform into "- Revenue", to be able to run the algorithm that only works for minimization
- the contraint function in PSO alogorithm are as : g(x) >=0. So I will transform : D1+D2+D3 <= 150 in : g(x) = -D1-D2-D3 +150 >= 0

I fix the lower price to 0 and the maximum price to 1750. At 1750€ even the Demand 3 is below 1 seat so there is no interest to test a bigger price then 1750€

I am looking for the parameters swarmsize and number of iteration. First step : I fix the number of iteration to nbiter = 1000 and I increase the number of swarm and plot the cost function relatively to the swarmsize.

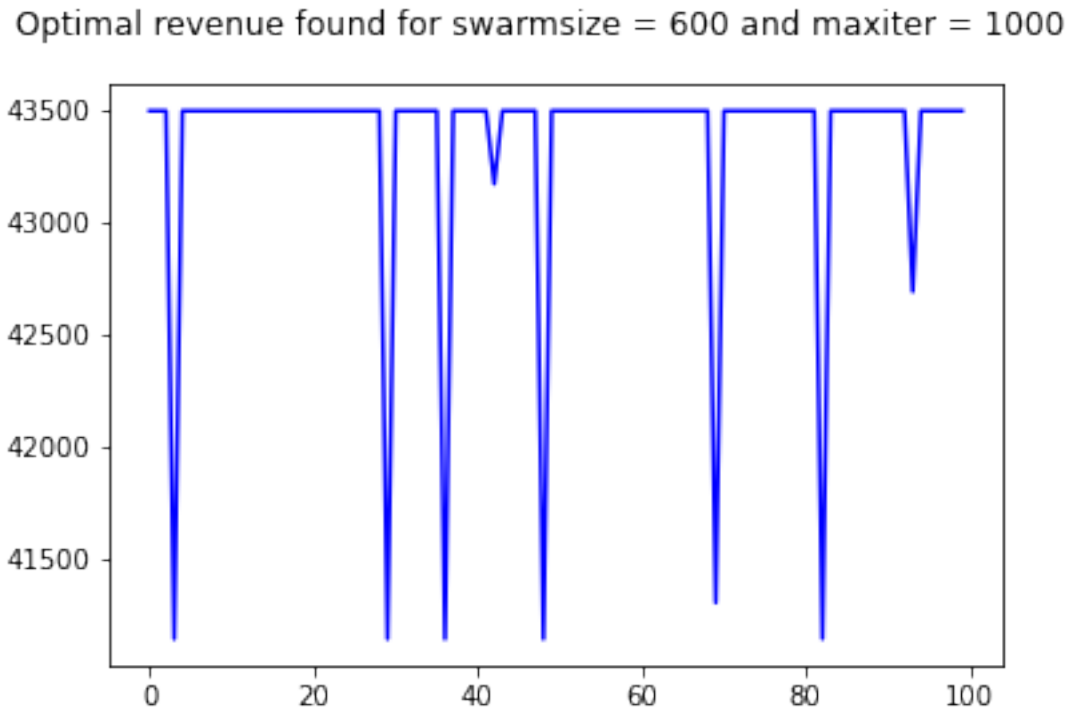
The graph shows that from a swarm size of 600 the finding of the minimum is stable Analysing the result of the pso function we also get the information that the process stops each time before reaching the full number of iteration so we can deduce that 1000 iterations are enough

```
Optimal Revenue 43494.68298993283, value of p1,p2,p3 [160.72246855 210.72215381
360.71290779], constraint : 150.00000000004607
```



1.2.4 Then I run PSO with swarmsize = 600 and iteration = 1000, 100 times to see if there are changes in the optimal result

```
Optimal Revenue : mean 43343.850187942975, max 41142.25950063026, min
43494.68298887, std dev 556.2854615962195
```



1.2.5 So the optimal revenue is 43494,68

I know try to found the best value so that the number of seats are integer values.

```
43554.10432213126 [160.9438 209.9576 357.8769] 20 37 91 148
43610.33886588461 [160.9438 209.9576 354.5982] 20 37 92 149
43612.05292098074 [160.9438 205.9574 357.8769] 20 38 91 149
43668.287464734094 [160.9438 205.9574 354.5982] 20 38 92 150
43612.588768618865 [156.0648 209.9576 357.8769] 21 37 91 149
43668.82331237222 [156.0648 209.9576 354.5982] 21 37 92 150
43670.53736746835 [156.0648 205.9574 357.8769] 21 38 91 150
43726.7719112217 [156.0648 205.9574 354.5982] 21 38 92 151
(156.06477482646685, 205.95737015548048, 354.59816928214826)
```

So we see that the optimal revenue with interger values for seats number and with number of seats <= 150 is :

- **cat1 : 21 seats (156,06€), cat2 : 38 sets(205,96€), cat3: 91 seats (357,88€) and a Revenue of 43670,54 €**

1.3 Sensitivity analysis

remove 3 seats <=> number maximum of seats = 147 seats. As a first approximation we could say that using the previous result a good decision would be to remove the three seats from cat 1 which is the less profitable categorie. Doing so the result would be :

- **cat1 : 18 seats (171,48€- new price related to D1), cat2 : 38 sets(205,96€), cat3: 91 seats (357,88€) and a Revenue of 43479 €**

But when running the simulation again with a constraint to 147 seats max, another optimum is found with the following result :

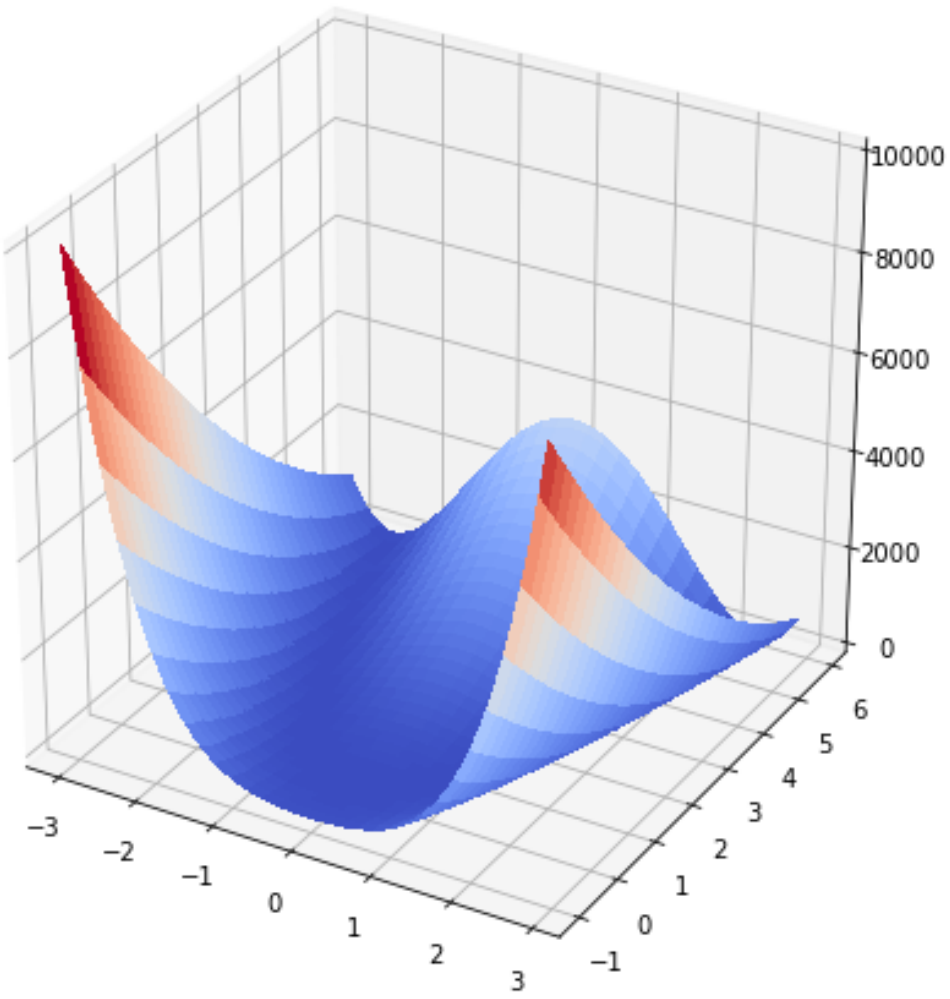
- **cat1 : 20 seats (160,94€), cat2 : 37 sets(209,96€€), cat3: 90 seats (361,19€€) and a Revenue of 43494,57€**

```
43432.56986212527 [161. 214. 361.] 20 36 90 146
43492.10117552927 [161. 214. 358.] 20 36 91 147
43494.57300872725 [161. 210. 361.] 20 37 90 147
43554.10432213126 [161. 210. 358.] 20 37 91 148
43491.05430861287 [156. 214. 361.] 21 36 90 147
43550.58562201688 [156. 214. 358.] 21 36 91 148
43553.057455214854 [156. 210. 361.] 21 37 90 148
43612.588768618865 [156. 210. 358.] 21 37 91 149
(156.06477482646685, 209.95760721780468, 357.8768904418053)
```

2 Ex2. Banana Function - with Gradient

Minimize f(x) = 100 * (x2-x1^2)^2 + (1-x1)^2, x1 and x2 belongs to K = [-5,5]

2.0.1 First, have a look at the function



2.0.2 Minimum ?

- K <> empty set, and K is closed
- f is continuous and infinite at infinity So it exists at least 1 minimum on K

But looking at the first graph we are sure that the function is not convex

So no insurance regarding the convergence of the gradient method and no real information regarding optimal values for the rho parameter of the gradient.

2.0.3 Minimize the function using a gradient method

As asked, I will use a gradient method that I progam in Python below : The fix step Gradient Method

For this, calculation of gradient :

* dJ/dx1=-400x1(x2-x1^2) - 2 (1-x1)
* dJ/dx2= 200(x2-x1^2)

There is no "real" constraint but the search univers is limited to (-5,5) for each variables.

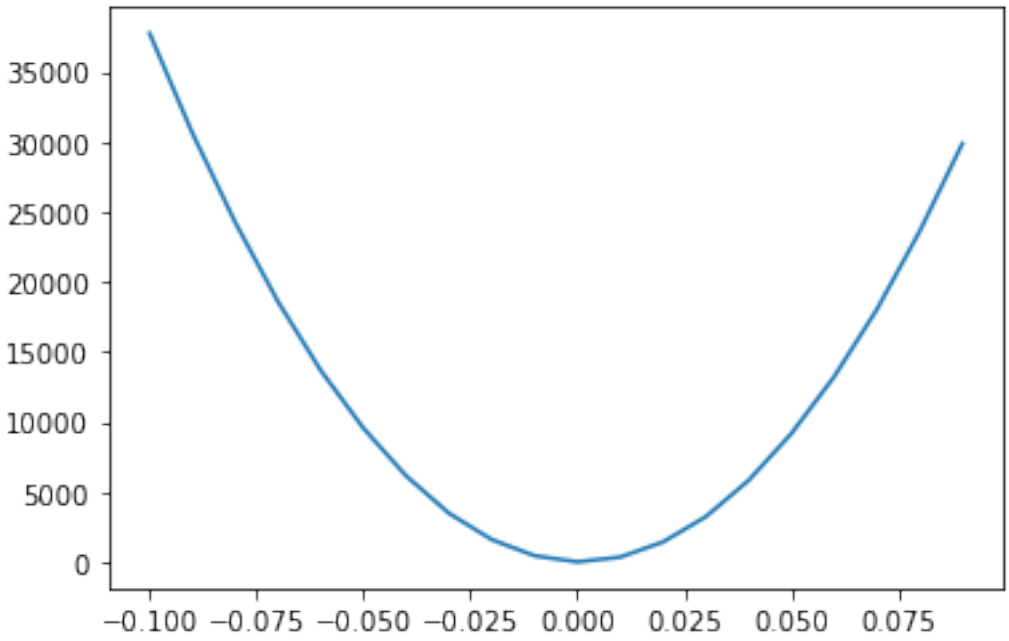
I choose to take in account this restricted universe using a "clap" function that realize a projection on the search universe when the design variables goes out of this search universe.

2.0.4 Process I followed

1) I first start applying the gradient method with different starting point. I rapidly understand that the "good" Rho parameter depends a lot on the starting position and that it was also the case for the number of iteration to realize to achieve a good convergence.

In particular if rho is too big : the algorithm do not converges and the x values get trapped in the border of the search universe, if rho is too small the algorithm is very slow to converge.

2) So I decided to improve my algorithm and to go for a "gradient optimal step" method in order to adjust the Rho parameter to the step values. I give up this idea because the Rho parameter seems to converge to 0 which has no interest. I not fully understand this but I plot a graph of the function to optimize in Rho for a particular x value (see below) and it seems to confirm this.



3) So I decide to optimize my alogrithm by calling the fix step gradient with successiv decreasing values of Rho, till the Gradient converge.

This is my final step

2.0.5 Implementation of my "fix step Gradient called with decreasing values of Rho"

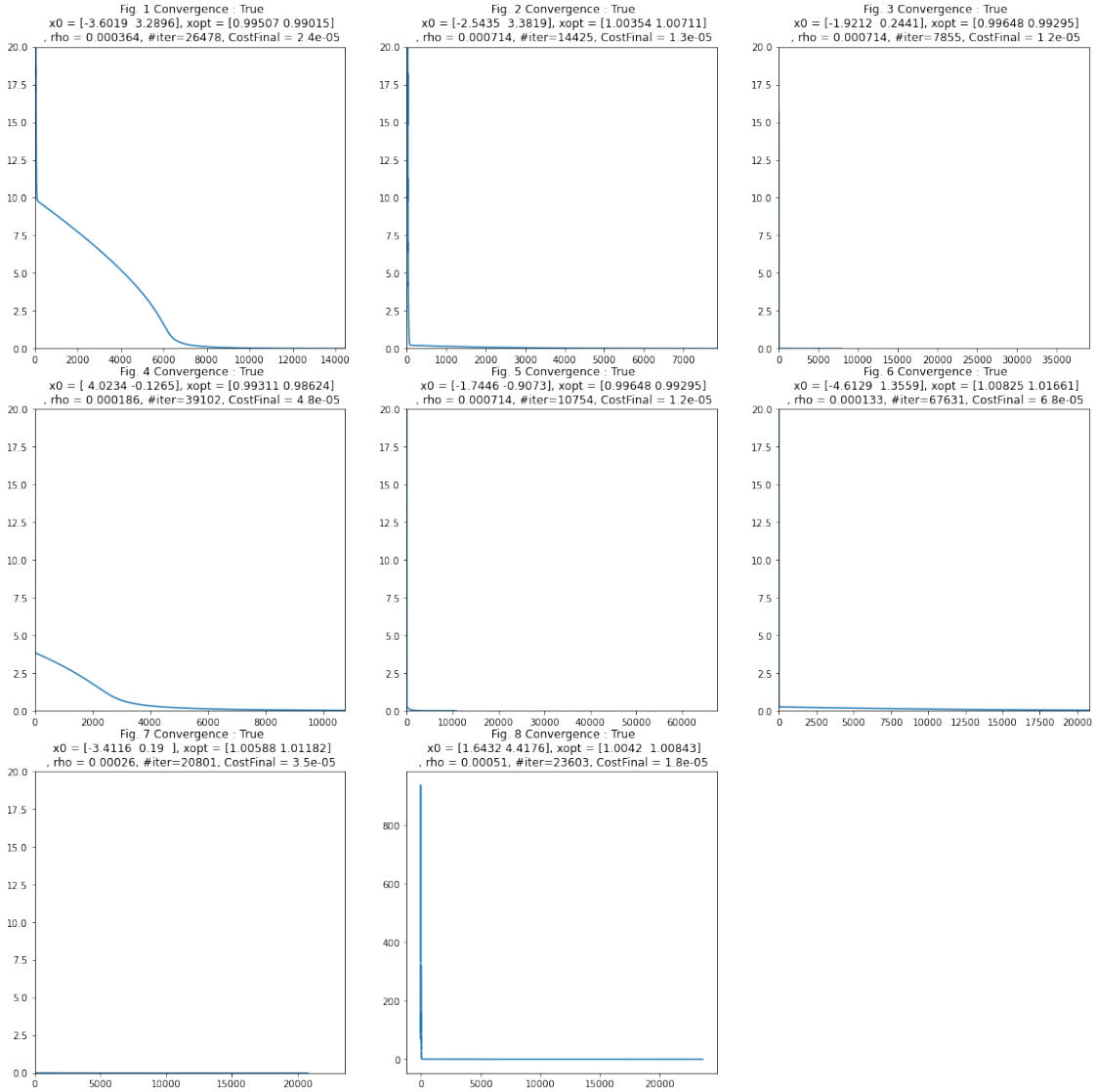
My idea was to pick randomly 1000 points in the search universe and to try to tune my algorithm so that it converges for most of the points and to analyse the datas given by this algorithm.

I spent quite a lot of time to tune this algorithm in particular to find the right :

- stop criteria : I finally choose the "cost function" stability with a tolerance of 1e-08
- initial rho value : I found that above Rho = 0.001 the Gradient almost never converge whatever is the starting point
- the number of time the Rho parameter is decreased
- factor of decrease for Rho. I found that 1.4 is a good trade-off
- maximum number of iteration before giving up : I found that 140000 enables to find most optimum values

2.0.6 Convergence graph for 9 starting point choosen randomly over 1000 test runs

I have plotted above 9 graph with objective value on y-axis and iteration number on x-axis



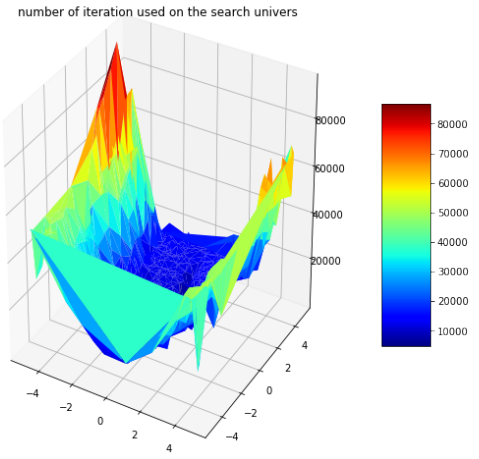
2.0.7 Result of the 1000 sample starting points test convergence

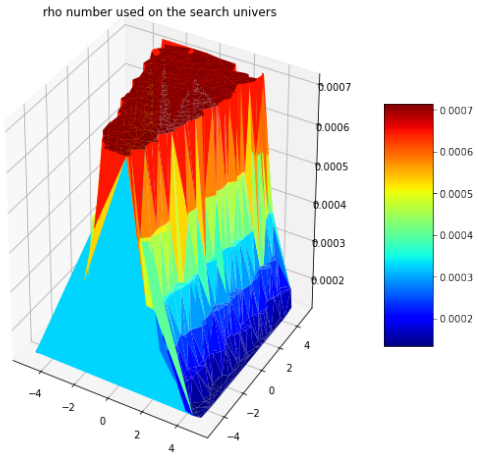
We can see that the convergence works for the whole 1000 starting points.

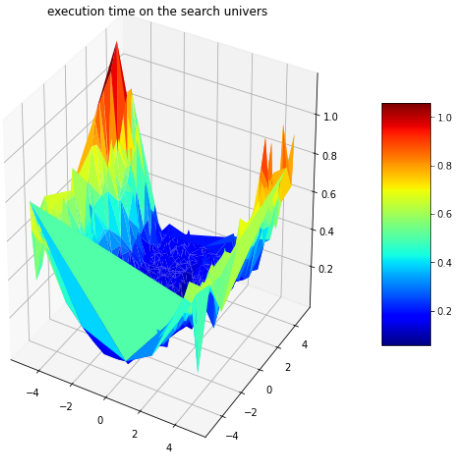
```
Convergence : 1000.0 convergences over 1000
```

```
Rho :            minimum 0.000133,      maximum 0.000714,      mean 0.000474
Iter. number :   minimum 796.0,         maximum 102057.0,      mean 24577.959
Obj. value :     minimum 1.2e-05,       maximum 6.8e-05,       mean 2.7e-05
Exec. time. :    minimum 11.7ms,        maximum 1534.0ms,      mean 338.3ms
```

I found also intersting to see where on the search universe the iteration number is high and where it is low, same for the execution time and the rho number found







It is interesting to notice here that the region "around" the x2-axis (x1 close to 0) needs low iteration, high rho number and a small execution time

3 Ex2. Eggcrate Function - Gradient method

Minimize f(x) = x1^2 + x2^2 + 25*((sin(x1))^2 + (sin(x2))^2), x1 and x2 belongs to K = [-2*PI*,2PI]

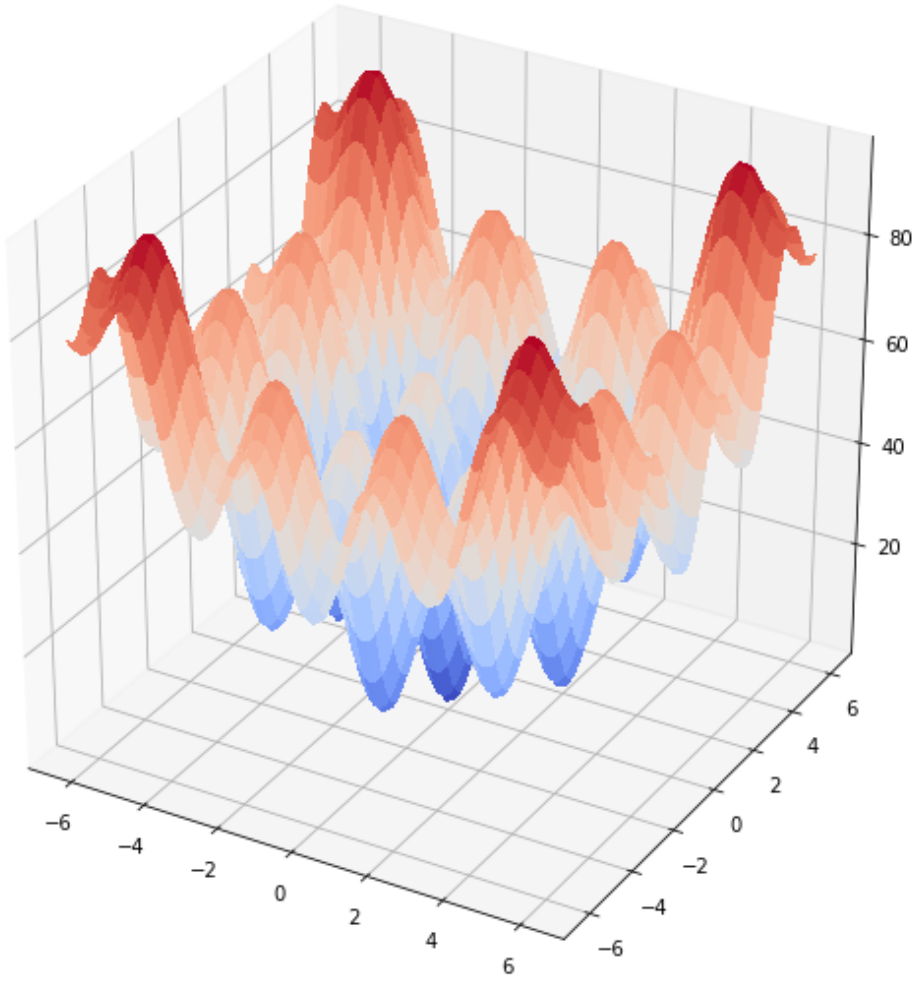
3.0.1 Minimum ?

- K <> empty set, and K is closed
- f is continuous and infinite at infinity So it exists at least 1 minimum on K

The convexity of f is not obvious

3.0.2 First let's have a look at the function

We can see that there are a lot of local optimal



A lot of local minimum means that if the starting point is far from the optimal point the global minimum can't be reached by a gradiet method. Only one of the local optimal will be reached.

3.0.3 Minimize the function using a gradient method

There is no "real" constraint but the search univers is limited to (-2*PI,2*PI) for each variables.

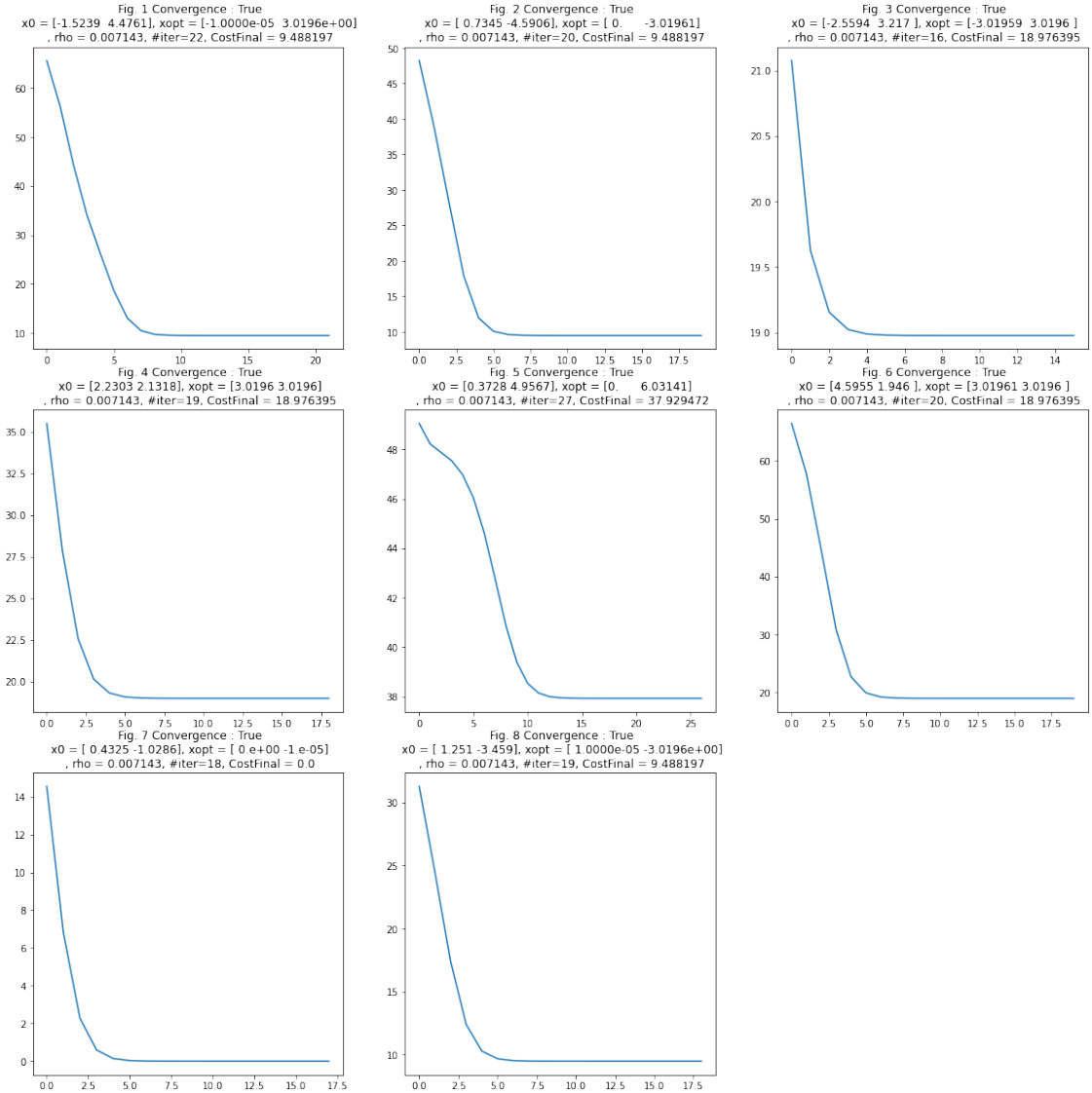
I choose to take in account this restricted universe using a "clap" function that realize a projection on the search universe when the design variables goes out of this search universe.

3.0.4 Process I follow

Here, in a way it is easier to apply "fix gradient method" because the same Rho can work every where on the search universe.

But what is interesting to find and to show is that the global optima can't be reach form all the points of the search universe.

To show this I use the same functions as before with Banana function but I will try to find different optimal (local and global)



3.0.5 9 sample convergence graph for 9 different starting point

I have plotted above 9 random graph with objective value on y-axis and iteration number on x-axis

We can see in this random sample that the algorithm converges several times to local optimum and only 1 time to the global optimal (fig 3)

3.0.6 Result of the 1000 sample starting points test convergence

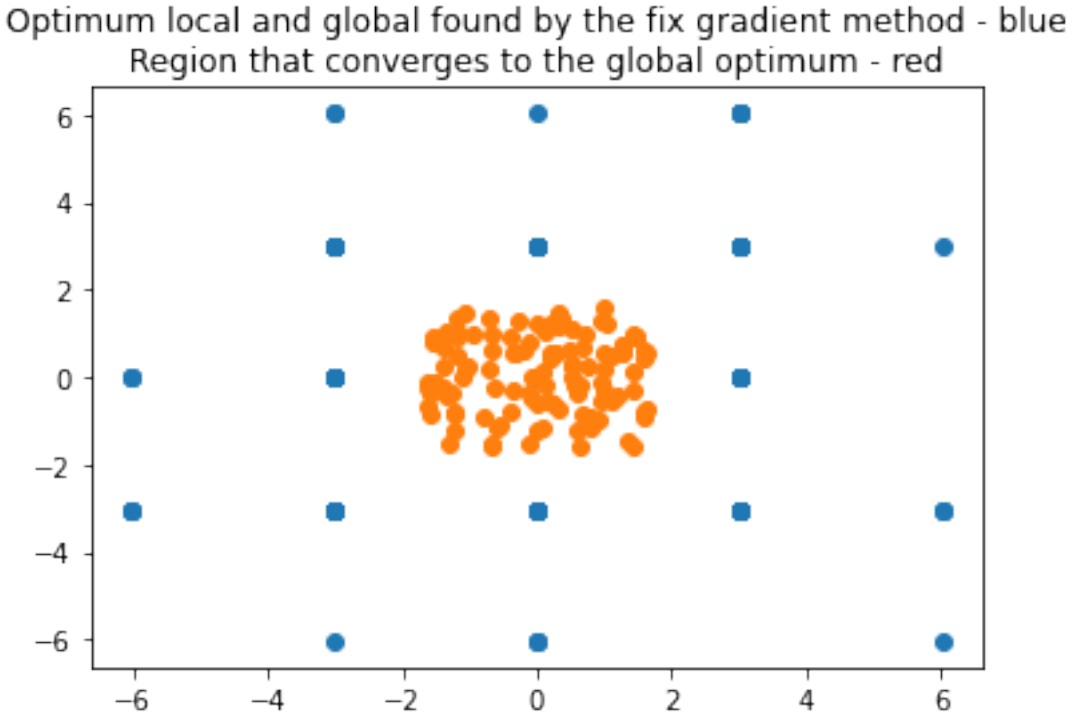
We can see that the convergence works for the whole 1000 starting points and that for some starting point the global minimum was found (objectiv value : minimum = 0.0)

```
Convergence : 1000.0 convergences over 1000
```

```
Rho :            minimum 0.007143,       maximum 0.007143,       mean 0.007143
Iter. number :   minimum 12.0,           maximum 45.0,           mean 19.544
Obj. value :     minimum 0.0,            maximum 75.858944,      mean 14.202899
Exec. time. :    minimum 0.0ms,          maximum 2.994ms,        mean 0.2ms
```

With EggCrate function, the "fix step gradient" method is simple (always the same rho used for all the points) and quick (iteration between 10 and 34), BUT the global optimum is only reached when the statring point is closed to the optimal point

The following graph shows the various optimal points reached by the algorithm and the region that converges to the global optimum



3.0.7 It's finally interesting to see the % of starting points thats converges to optimal minimum

```
% of starting point leading to global minium : 11.2 %
```

4 Ex2. Golinski's Speed Reducer - Gradient method

Here the function to minimize is 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)

This is a complex function with 7 variables and also 11 constraint functions given in the documentation.

My idea is to try to use the Uzawa's algorithm because this algorithm combine gradient methode and ability to deals with constraints.

4.0.1 Process I follow

Understanding the complexity of the function and of the constraints, it is highly probable that this function has lots of minimum.

The first think was to calculate the different function needed : Gradient of the cost function and Gradient of the Lagrangien for x and for lambda

Not easy calculation !

I choose to take in account the restricted universe using a "clap" function that realize a projection on the search universe when the design variables goes out of this search universe. The same for the lambda vector when it becomes negative

I first test my calculation for gradient and Lagrangien with numeric calculation to be sure that there were no calculation errors.

For instance the test of gradient below, compare the derivative I calculate to a numeric calculation : And it is quit good :

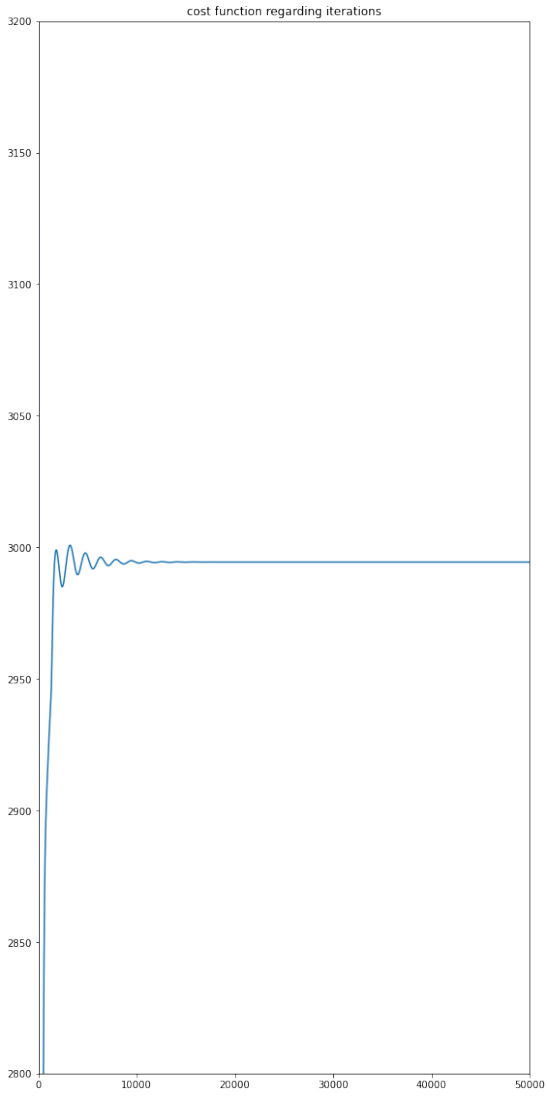
```
gradient function [ 534.05982065 4627.03605771  167.92757379    7.25835264
21.07416696
  218.51940128  620.18400556]
numeric controle [ 534.05981816 4627.03636913  167.92757833    7.2583498
21.07416549
  218.51940346  620.18401877]
```

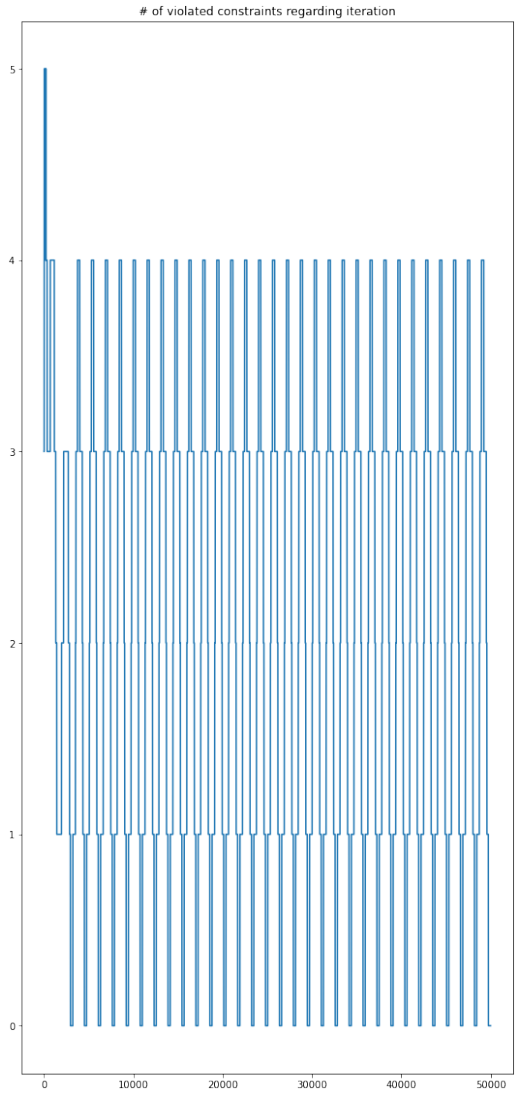
I try different rho1 and rho2 parameters for the alogorythm. Rho1 is for the solution gradient and rho2 is for the lambda optimization. A good compromise is rho1 = 0.0001 and lambda = 10.

I run below the algorithm, with a number of iteration of 50000 and a random starting point

```
 minimum cost : 2994.3550261330092,
 optimal position : [ 3.5         0.7        17.          7.3         7.71531991
3.35021467
  5.28665446]
 values of the constraint : [-7.39152804e-02 -1.97998527e-01 -4.99172248e-01
-9.04643905e-01
 -1.40998324e-14 -6.91668944e-14 -7.02500000e-01 -5.63660230e-13
 -5.83333333e-01 -5.13257535e-02 -1.12786447e-10]

 execution time : 9.637961387634277 s
We can check that all the values of the constraint vector are negative
```

[369]: 2941





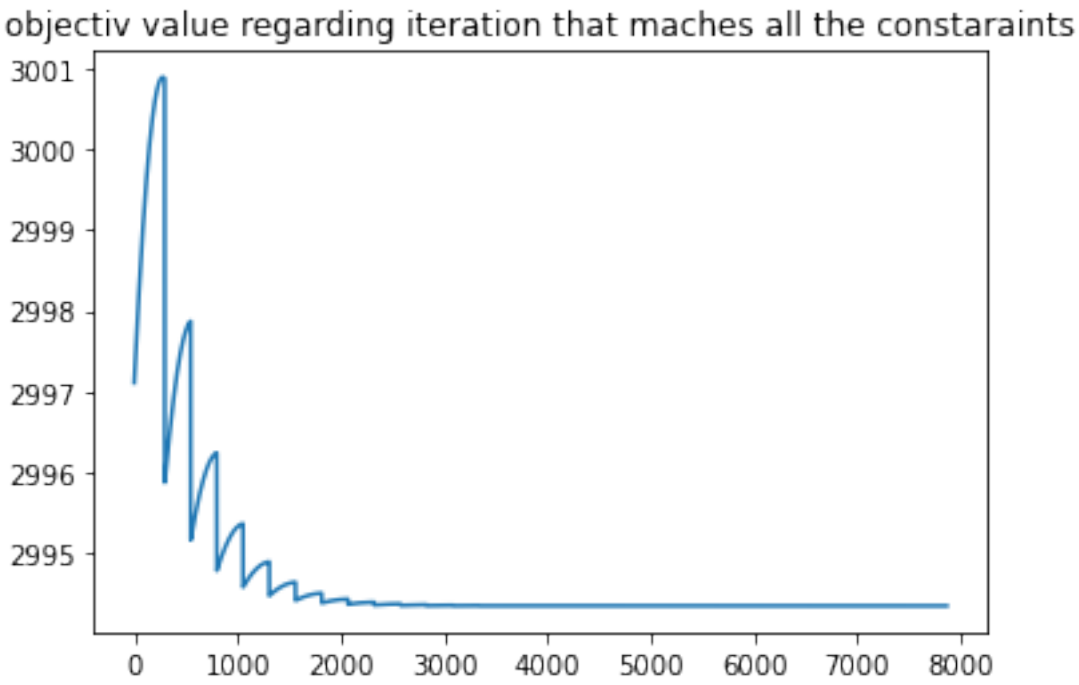
The graph shows that :

- the cost converges to a very low value : 2994.355026133452
- but the number of constraint violation oscillate between 0 and 4 which means that the minimization is quit instable in terme of constraint violation.

I need to have a rho2 quit high (10) other wise the algorithm do not take in account the constraints and converges with a very low cost function but with a very high number of constraints. But I think that the instability of the solution (regarding constraint violation) is a consequence of this high rho2.

A possible improvment of the algorythm could be to decrease rho2 after a certain time.

It's interesting to have a look at the values of the objectiv function for which the constraints are satisfied



5 Ex 3. Banana function - Heuristic technics

I've decided to use the same PSO algorithm that I use and slightly modify in the first exercice

The PSO algorithm is adapted to continuous variable, quick and start from several starting point instead of just one in gradient method. So I think that it should be adapted to the "Banana function"

After a few tests, it's not a surpise to see that the algo found the minimum "quite easily" but it's interesting to see now the impact of the different parameters of the algorithm and the way to tune these parameters in order to find an optimized execution.

These are the parameters that can be given to the algorithm

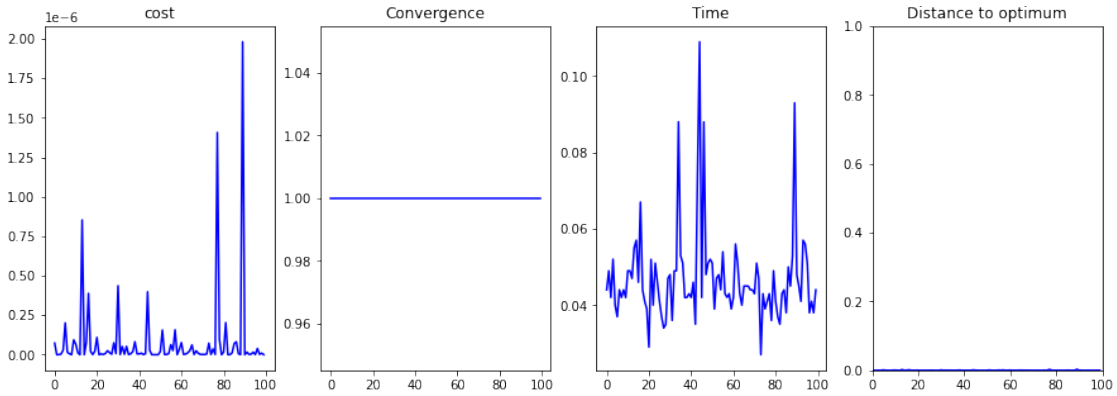
- swarmsize : The number of particles in the swarm (Default: 100)
- omega : Particle velocity scaling factor (Default: 0.5)
- phip : Scaling factor to search away from the particle's best known position (Default: 0.5)
- phig : Scaling factor to search away from the swarm's best known position (Default: 0.5)
- maxiter : The maximum number of iterations for the swarm to search (Default: 100)
- minstep : The minimum stepsize of swarm's best position before the search terminates (Default: 1e-8)
- minfunc : The minimum change of swarm's best objective value before the search terminates (Default: 1e-8)

The idea is to first try the algorithm with the default values, and then to try to adjust the parameters to see if better results can be achieved. For this, the algorithm is run 100 times to have an average vision of the result and the four following graphes are plotted :

- cost function for the 100 runs
- convergence status for the 100 runs
- time to find convergence for the 100 runs
- distance to the optimal point(1,1) for the 100 runs

This first try give 100% of convergence to minimum and an execution time around 46ms

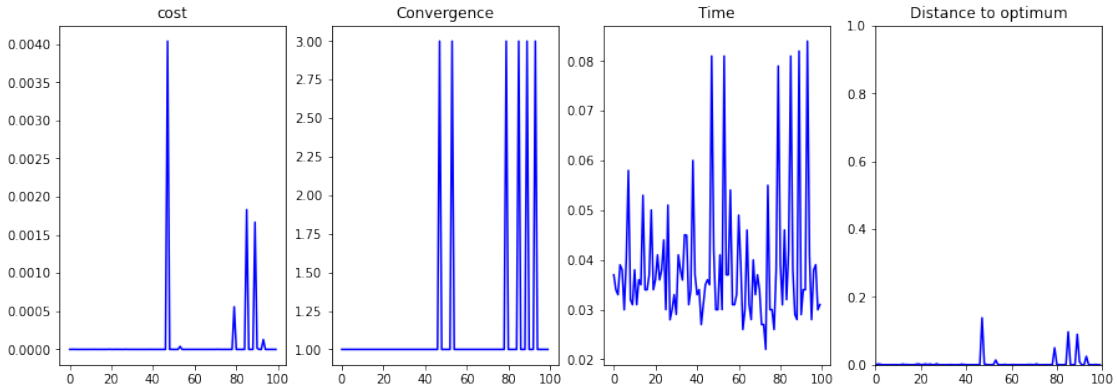
```
distance to optimal point <0.1 : 100 / 100,  mean time exec : 46.92033767700195
ms
```



Decreasing swarmsize by step of 10 shows that 70 is a good compromise

- % of distance to optimal point is > 98% smaller then 0.1
- the execution time decrease down to around 38ms

```
distance to optimal point <0.1 : 99 / 100,  mean time exec : 38.68971347808838
ms
```



Another idea is to change the "swam parameters"

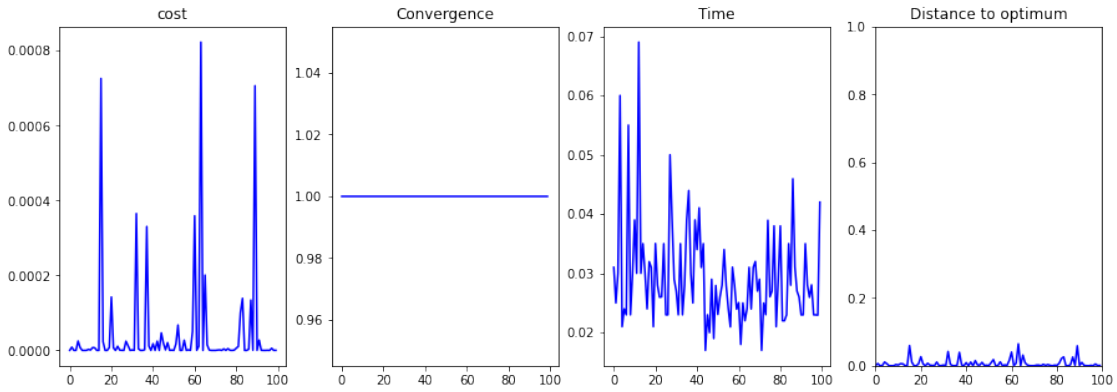
- omega : Particle velocity scaling factor (Default: 0.5)
- phip : Scaling factor to search away from the particle's best known position (Default: 0.5)

- phig : Scaling factor to search away from the swarm's best known position (Default: 0.5)

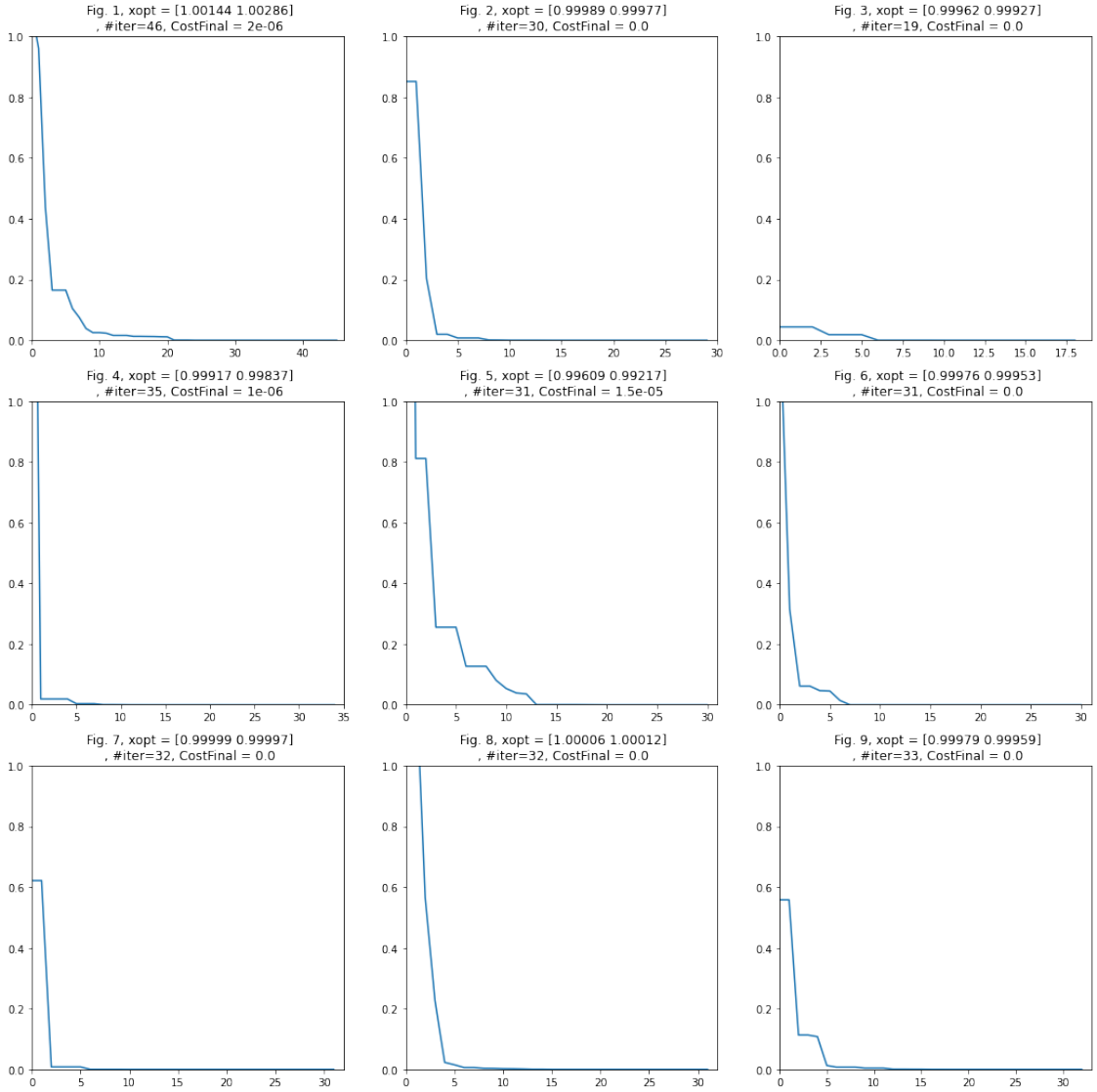
My searches on those 3 parameters shows that a good compromise is : omega=0.3, phip=0.99 and phig=0.5

The quality of the found optimal is still good and the execution time goes down around 30ms

`distance to optimal point <0.1 : 100 / 100,  mean time exec : 29.38997983932495 ms`



9 plotting with cost function regarding iteration runned with the final parameters.



6 Ex 3. Eggcrate function - Heuristic technics

I've decided to use the same PSO algorithm used in the previous exercices

The PSO algorithm is adapted to continuous variable, quick and start from several starting point instead of just one in gradient method. So I think that it should be adapted to the "Eggcrate function" and able to find global optimum rather than local optimum

After a few tests, it's not a surpise to see that the PSO found the global minimum "quite easily" but it's interesting to see now the impact of the different parameters of the algorythm and the way to tune this parameter in order to find an optimized execution.

These are the parameters that can be given to the algorithm

- swarmsize : The number of particles in the swarm (Default: 100)

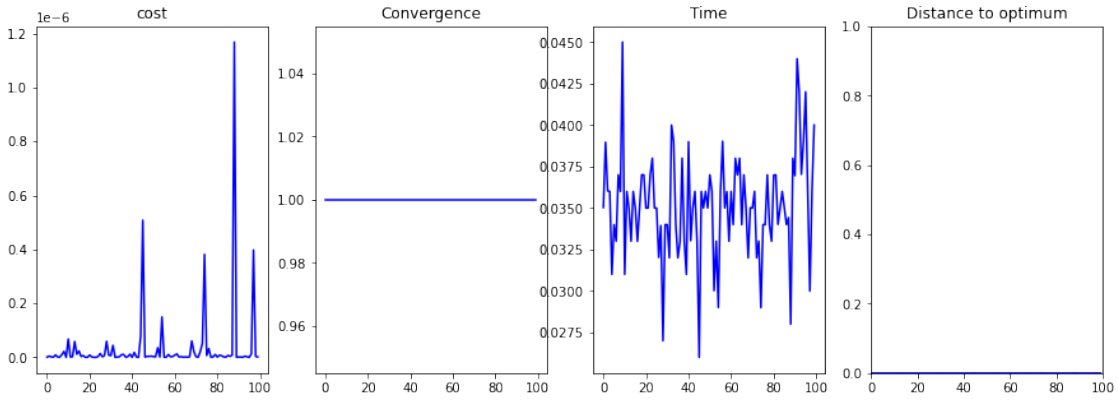
- omega : Particle velocity scaling factor (Default: 0.5)
- phip : Scaling factor to search away from the particle's best known position (Default: 0.5)
- phig : Scaling factor to search away from the swarm's best known position (Default: 0.5)
- maxiter : The maximum number of iterations for the swarm to search (Default: 100)
- minstep : The minimum stepsize of swarm's best position before the search terminates (Default: 1e-8)
- minfunc : The minimum change of swarm's best objective value before the search terminates (Default: 1e-8)

The idea is to first try the algorithm with the default values, and then to try to adjust the parameters to see if better results can be achieved. For this, the algorithm is run 100 times to have an average vision of the result and the four following graphes are plotted :

- cost function for the 100 runs
- convergence status for the 100 runs
- time to find convergence for the 100 runs
- distance to the optimal point(1,1) for the 100 runs

This first try give 100% of convergence to minimum and an execution time around 35ms

```
distance to optimal point <0.1 : 100 / 100,  mean time exec : 35.10459899902344
ms
```



I decrease the swarmsize in a loop function and print the result of each 100 runs to find the best swarmsize

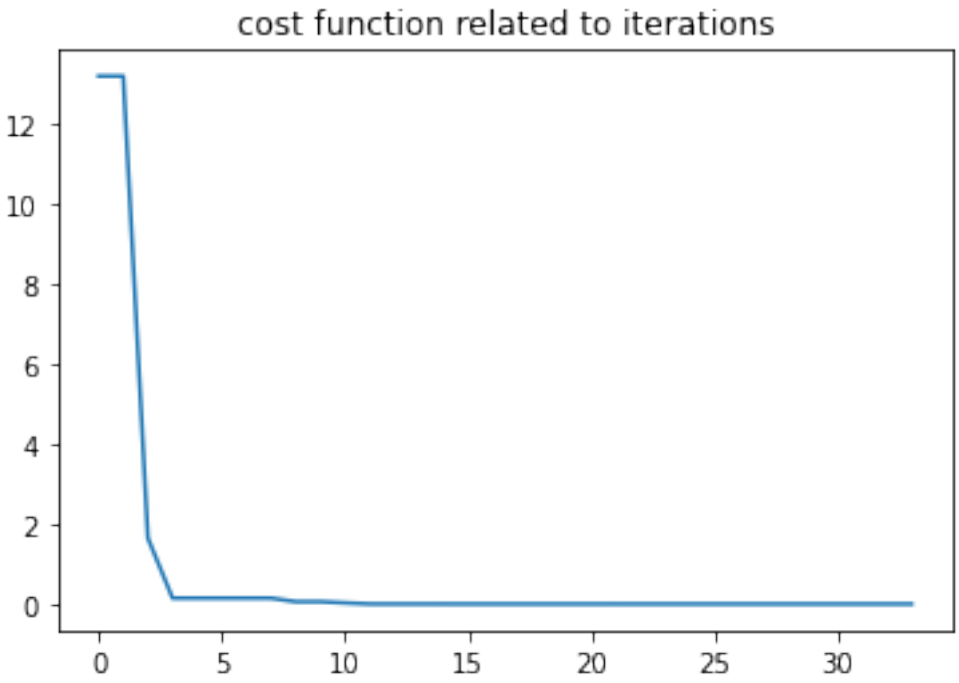
```
# swarm : 100, % convergence : 100 / 100,  max Cost : 0.0, time = 34.0
# swarm : 96, % convergence : 100 / 100,  max Cost : 0.0, time = 33.0
# swarm : 92, % convergence : 100 / 100,  max Cost : 0.0, time = 36.0
# swarm : 88, % convergence : 100 / 100,  max Cost : 0.0, time = 34.0
# swarm : 84, % convergence : 100 / 100,  max Cost : 0.0, time = 32.0
# swarm : 80, % convergence : 100 / 100,  max Cost : 0.0, time = 31.0
# swarm : 76, % convergence : 100 / 100,  max Cost : 0.0, time = 28.0
# swarm : 72, % convergence : 100 / 100,  max Cost : 0.0, time = 27.0
# swarm : 68, % convergence : 100 / 100,  max Cost : 0.0, time = 26.0
# swarm : 64, % convergence : 100 / 100,  max Cost : 0.0, time = 25.0
```

```
# swarm : 60, % convergence : 100 / 100,  max Cost : 0.0, time = 23.0
# swarm : 56, % convergence : 100 / 100,  max Cost : 0.0, time = 22.0
# swarm : 52, % convergence : 100 / 100,  max Cost : 0.0, time = 20.0
# swarm : 48, % convergence : 98 / 100,  max Cost : 9.49, time = 19.0
# swarm : 44, % convergence : 100 / 100,  max Cost : 0.0, time = 17.0
# swarm : 40, % convergence : 100 / 100,  max Cost : 0.0, time = 17.0
# swarm : 36, % convergence : 100 / 100,  max Cost : 0.0, time = 15.0
# swarm : 32, % convergence : 100 / 100,  max Cost : 0.0, time = 14.0
# swarm : 28, % convergence : 99 / 100,  max Cost : 9.49, time = 12.0
# swarm : 24, % convergence : 98 / 100,  max Cost : 9.49, time = 11.0
# swarm : 20, % convergence : 96 / 100,  max Cost : 9.49, time = 9.0
# swarm : 16, % convergence : 91 / 100,  max Cost : 9.49, time = 8.0
# swarm : 12, % convergence : 79 / 100,  max Cost : 9.49, time = 7.0
# swarm : 8, % convergence : 74 / 100,  max Cost : 9.49, time = 6.0
# swarm : 4, % convergence : 45 / 100,  max Cost : 39.41, time = 6.0
```

30 seems to be a good compromise for this swarmsize

In that case the execution time is around 15ms

At that time it is interesting to have a look at the graph of the cost function related to the number of iteration



We can see that the cost function is decreasing very rapidly, so we can imagine to be able to improve the execution time by modifyin the criteria of the stop function on the objective function

So I decrease the minfunc parameter in a loop function and print the result of each

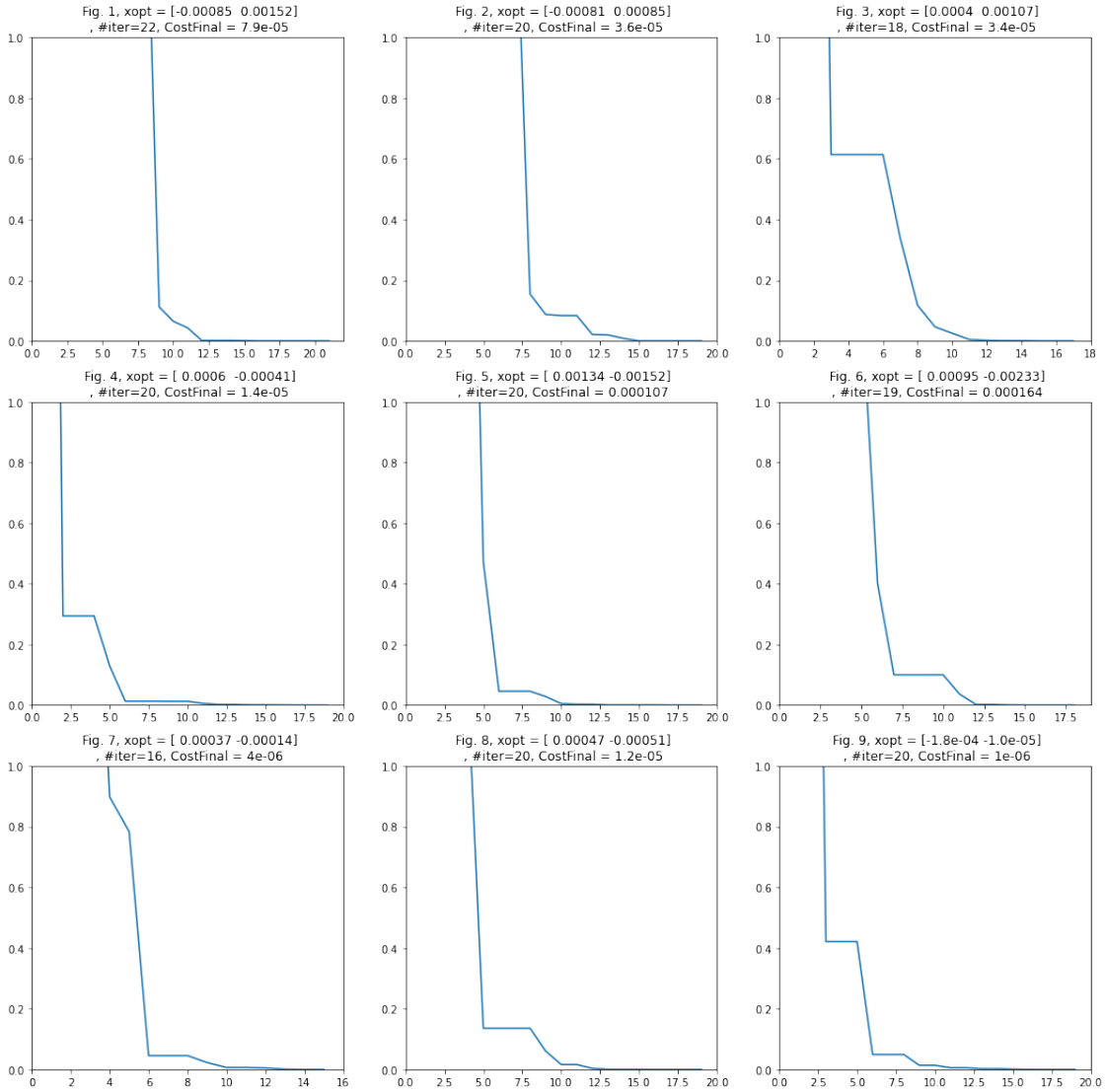
100 runs to find the best minfunc

```
# minfunc : 1e-08, % convergence : 99 / 100,  max Cost: 9.49, time = 13.0
# minfunc : 1e-07, % convergence : 97 / 100,  max Cost: 9.49, time = 11.0
# minfunc : 1e-06, % convergence : 99 / 100,  max Cost: 9.49, time = 10.0
# minfunc : 1e-05, % convergence : 96 / 100,  max Cost: 9.49, time = 9.0
# minfunc : 0.0001, % convergence : 98 / 100,  max Cost: 9.49, time = 8.0
# minfunc : 0.001, % convergence : 98 / 100,  max Cost: 9.49, time = 7.0
# minfunc : 0.01, % convergence : 98 / 100,  max Cost: 0.31, time = 6.0
# minfunc : 0.1, % convergence : 83 / 100,  max Cost: 14.71, time = 4.0
```

Results are good until minifunc = 1e-4

The execution time become close to 8 ms

with those parameters fixed, I plot 9 graph, choosed randomly, showing cost function related to number of iteration



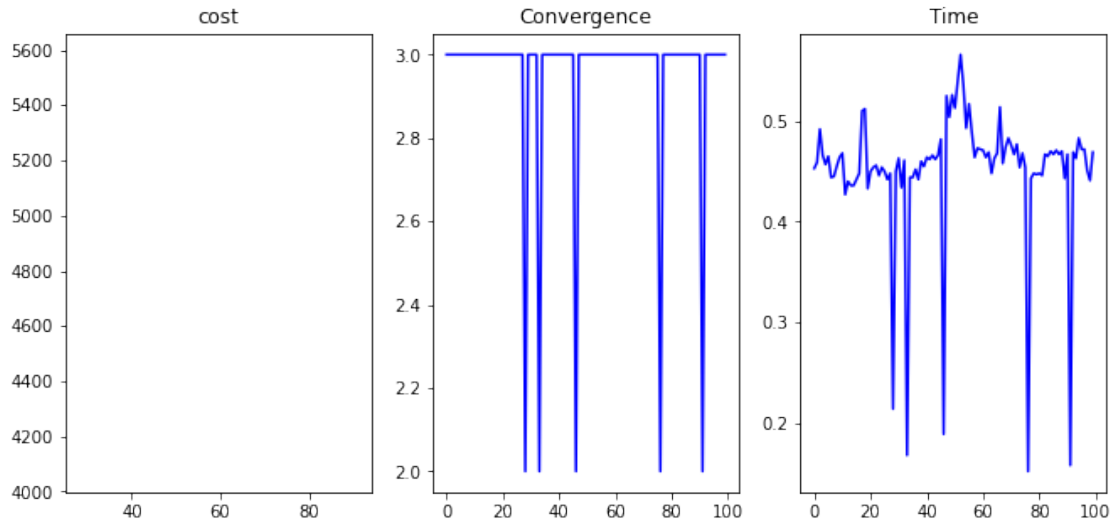
7 EX 3. Golinski's with PSO - Heuristic Technics

The idea is to run the PSO algorithm on Golinski speed reducer.

The implementation of this algorithm enable to define a constraint function and also boundaries for the variables. So it seems quit wall adapted to our case.

As for the previous exemple the idea is to first test what happens with the standard parameters of PSO

```
minimum cost 4073.4162369180576,  mean time exec : 452.04994678497314 ms
```



The result is not good, none of the 100 test manage to find a solution that match the constraints

As the search space is very complex, I decide to increase the swarmsize step by step to see if doing this the system start findind minimum solution.

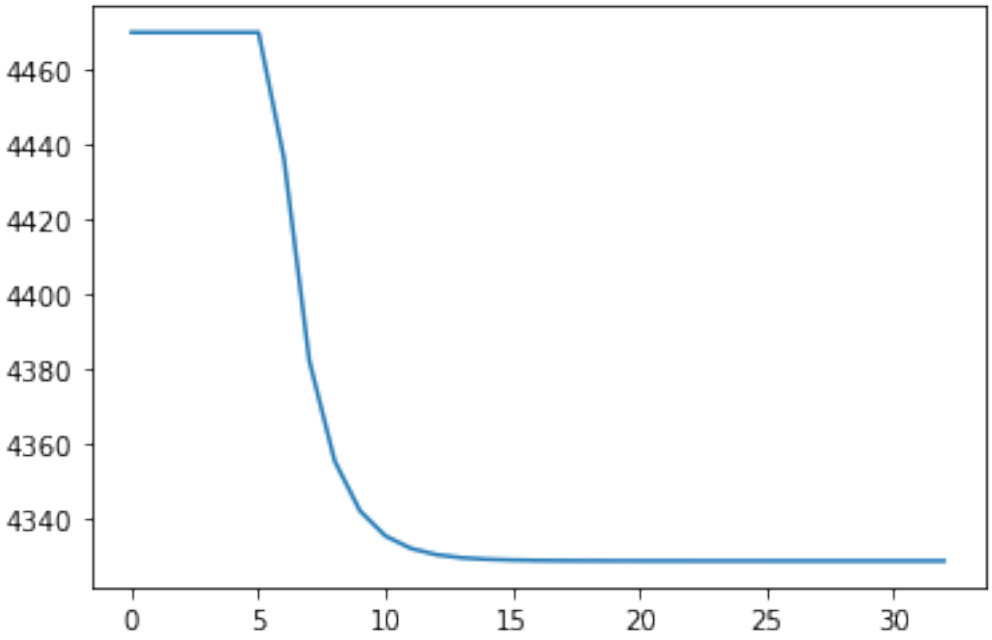
```
# swarm : 100, # cost < 5000 : 1 / 10, min Cost : 3372.67, time = 422.0 ms
# swarm : 200, # cost < 5000 : 2 / 10, min Cost : 3483.96, time = 834.0 ms
# swarm : 400, # cost < 5000 : 5 / 10, min Cost : 3232.18, time = 1196.0 ms
# swarm : 800, # cost < 5000 : 4 / 10, min Cost : 3209.85, time = 2135.0 ms
# swarm : 1600, # cost < 5000 : 6 / 10, min Cost : 3185.42, time = 4133.0 ms
# swarm : 3200, # cost < 5000 : 9 / 10, min Cost : 3152.57, time = 10418.0 ms
# swarm : 6400, # cost < 5000 : 10 / 10, min Cost : 3054.91, time = 19214.0 ms
```

We notice that the number of run that converge and the lowest cost value achieved improves when increasing the number of swarm.

As a compromise I choose to set the number of swarm to 5000

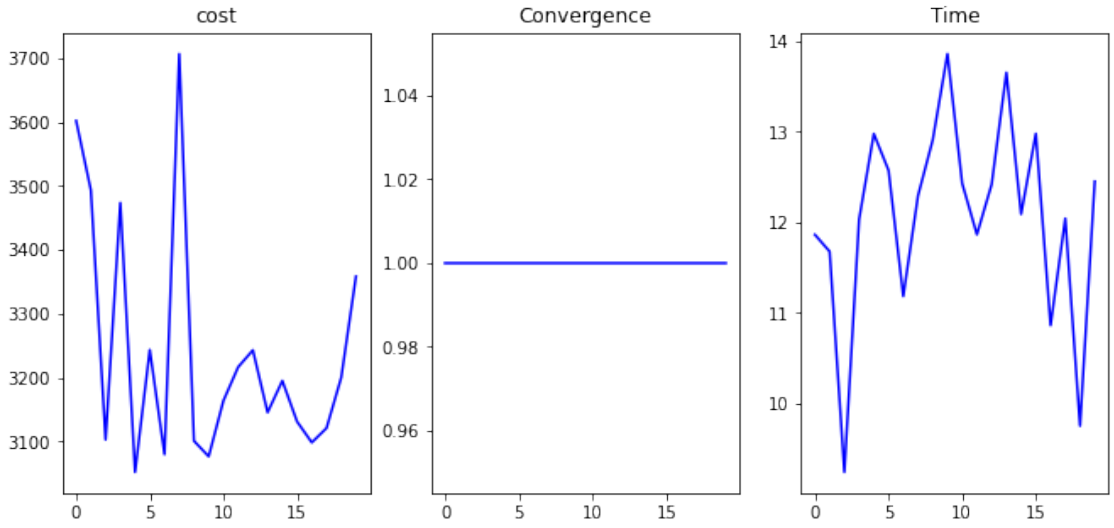
The execution times are quit long with a high number of swarm and we are far from the optimal cost found with the gradient method (around 2994), so I start looking at the other parameters.

First I plot below the cost regardind the number of iteration. We can see that is is going quickly down and then the minimum seems not to change a lot. That means that if I increase the stop criterion, I won't modify to much the optimum but I will have quicker execution.



So I try to run PSO (20 run) with swarm size = 5000, and stop croterion =1e-04 to see the result

`minimum cost 3052.4781715152726,  mean time exec : 12057.759535312653 ms`



Result is really better and quicker (12s instead of around 15s with the previous stop criterion)

But the cost found is still "high" regarding what I have found the gradient method.

So, the idea is to change the (phip,phig,omega) parameters to see which combinaison fit the better.

After several unitary test, the best solution are for a phip = 0, phig around 3 and not changing omega. (see below for the impact of phig).

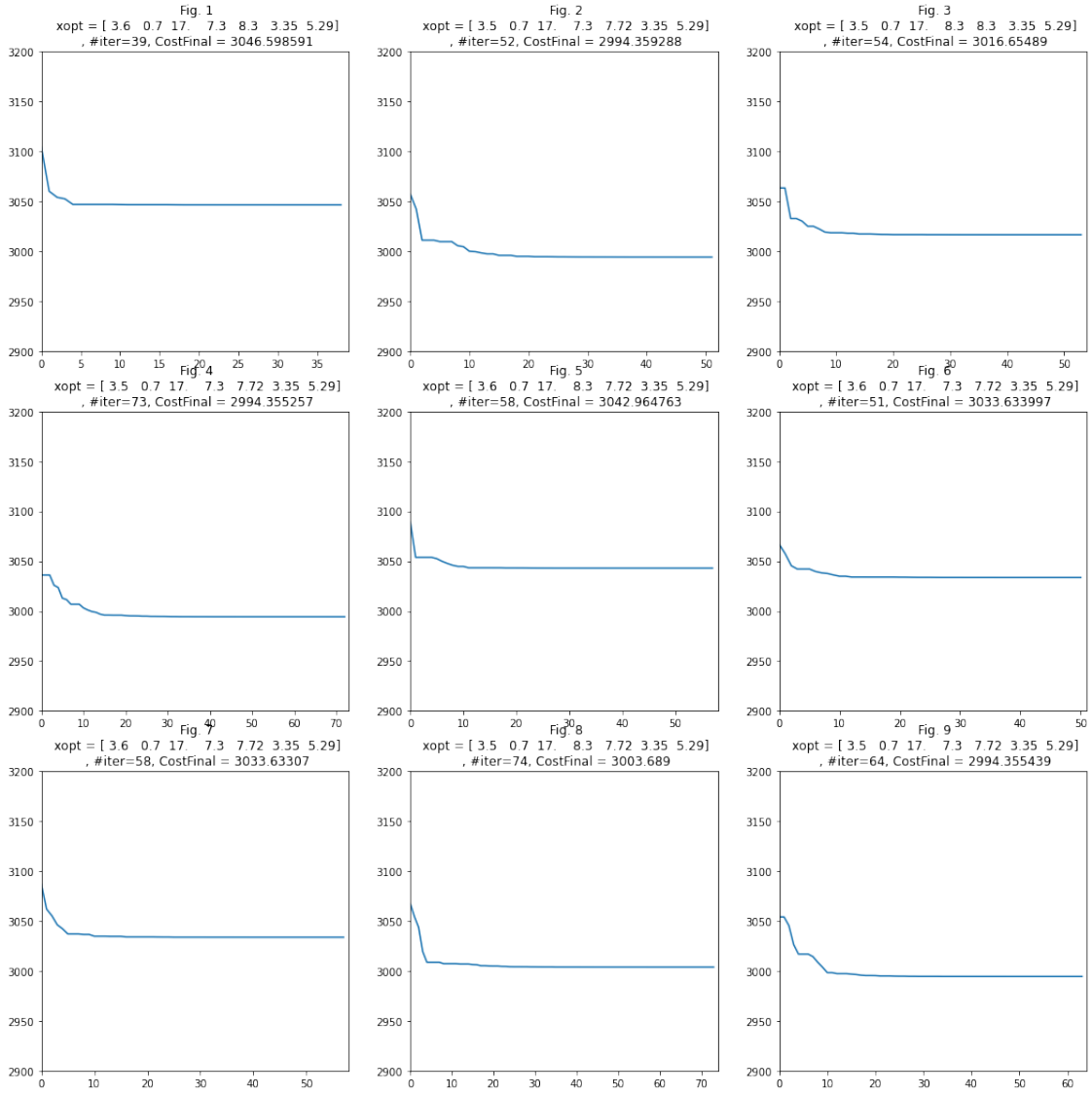
```
# phig : 0.1, # cost < 5000 : 5 / 5, min Cost : 3162.2, time = 8248.3264 ms
# phig : 0.6, # cost < 5000 : 5 / 5, min Cost : 3009.22, time = 13754.7893 ms
# phig : 1.1, # cost < 5000 : 5 / 5, min Cost : 2995.46, time = 17152.22507 ms
# phig : 1.6, # cost < 5000 : 5 / 5, min Cost : 2994.36, time = 18035.80637 ms
# phig : 2.1, # cost < 5000 : 5 / 5, min Cost : 2994.36, time = 14529.20017 ms
# phig : 2.6, # cost < 5000 : 5 / 5, min Cost : 2994.36, time = 17397.64872 ms
# phig : 3.1, # cost < 5000 : 5 / 5, min Cost : 2994.36, time = 20982.20224 ms
# phig : 3.6, # cost < 5000 : 5 / 5, min Cost : 2994.36, time = 22003.12743 ms
```

My final step is :

- to take the parameters that I have selected
- to run the PSO a high number of time
- to keep from all this runs the best result.

As the PSO algorythme is stochastic, running it a high number of time can lead to find better solutions

For 9 of this run, I plot the graph of cost regarding number of iteration.

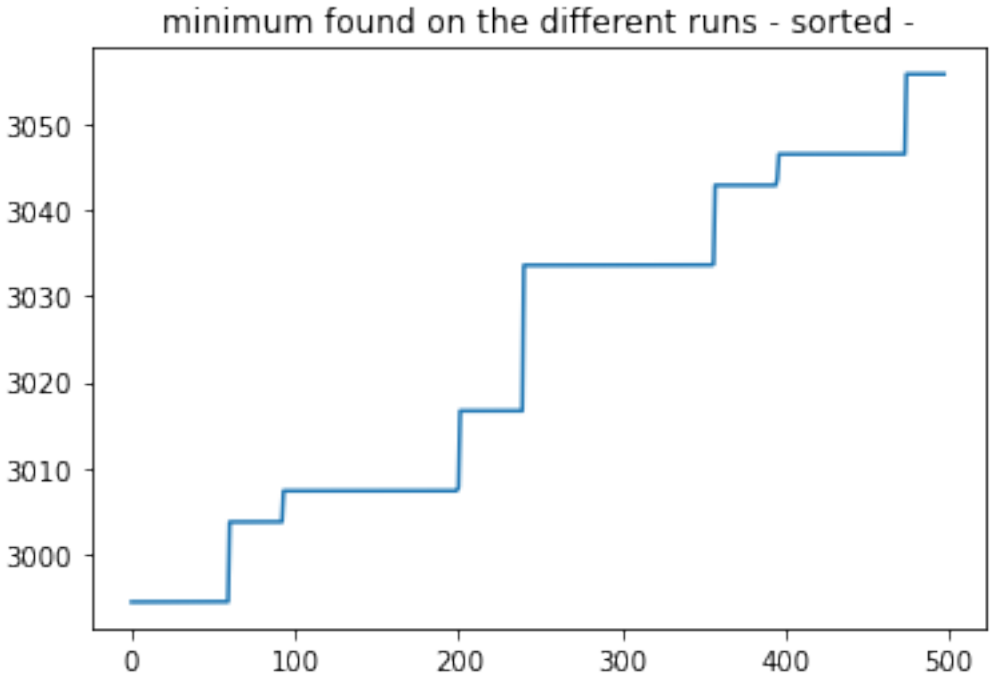



```
Best solution found [ 3.50000007  0.7        17.          7.3          7.71532274
3.35021473
  5.28665449],

 with objective value 2994.3551501857487          and

 the constraint vector is [-7.39152978e-02 -1.97998542e-01 -4.99172288e-01
-9.04643802e-01
 -6.02331316e-08 -1.64440734e-08 -7.02500000e-01 -1.88235512e-08
 -5.83333325e-01 -5.13257397e-02 -3.62070699e-07]
```

So for this run the best find solution has an objective value of 2994.3551501857487 and we can check that all the constraints are negatives. In previous test I reach at best an objective function of 2994.3550404629764 at position (

3.5,0.7,17.,7.3,7.71532031,3.35021468,5.28665447)

I finally found interesting to plot the different optimal cost found by the algorithm to show that it is as if there were several local minimum found.



8 General conclusion

8.1 Banana function

8.1.1 Dependance on initial design vector

Gradient method is very sensitive to the starting point for this function. Doing gradient method with fix step is I think impossible because of this.

This is a main advantage for this function for meta heuristic (PSO) since the algorithm compute several random starting points.

8.1.2 Computational effort

For Gradient, depending on the starting point it varies from 11ms to 1530ms in my tests

For PSO, I have a final computational effort which was around 30ms with almost no deviation.

So with no surprise for this function, PSO is much quicker.

8.1.3 Convergence history

For Gradient method, it depends a lot on the starting point

For PSO, the convergence is quick on all the tests a good approximation is reached after 10 iterations

8.1.4 Frequency at which the technique gets trapped in local optimum

I do not found local optimum for this function

But for gradient optimal the algorithm was often block in the boundaries of the search universe.

8.2 Eggcrate function

8.2.1 Dependance on initial design vector

Gradient method is very sensitive to the starting point for this function. Because depending on the starting point, you will reach either a local minimum or a global minimum.

The PSO can have the same issue if the number of swarm is really too small. But In my tests with a swarmsize of 30 the system almost converges to the global optimal at each run.

8.2.2 Computational effort

For Gradient, the algorithm is really quick : my tests gives an execution time in mean at 0.2ms (going up to 1ms maximum), but of course the optimal point is not found for each of these runs.

For PSO, I have a final computational effort which was around 10ms with almost no deviation.

So here gradient is quicker but often reach local minimum.

8.2.3 Convergence history

For Gradient, my step criteria stop the iterations around 20 steps, but we can see on the graph that the convergence is already good after 5 to 7 iterations

For PSO, the convergence is quick on all the tests a good approximation is reached after 15 iterations

8.2.4 Frequency at which the technique gets trapped in local optimum

The Frequency at which the technique gets trapped in local optimum is around 10% in my test.

As previously said with swarmsize = 30 the % of convergence on a local optimum is verys small

8.3 Golinski' speed reducer

8.3.1 Dependance on initial design vector

With the Uzawa methode I try several random start and the result has always been the same. It's a bit strange, but I do not find dependance on initial vector.

Same for PSO.

8.3.2 Computational effort

For Uzawa methode the execution time is around 10s.

For PSO, I have an execution time between 10 and 15s for each run. To get my best solution I run the same algorithme 500 times. This means a cumulative time of approximatively 2 hours.

8.3.3 Convergence history

For Uzawa the convergence at a good level needs around 15000 iterations

For PSO it is really less : around 10 or 20 iterations

At the end, even if the difference is very small, Uzawa found a smaller cost value.

8.3.4 Frequency at which the technique gets trapped in local optimum

With the parameters that I have found, the Uzawa never get trapped in a local minimum

Whereas for PSO, the percentage was quit high (around 90%) perhaps due to parameters not completly fitted.

8.3.5 Comparison with paper given with the assignment

I am a bit surprise that the cost values I found are smaller then the one in the paper (3004) and also better then the best one mention in the paper (2996 for Tapabrata Ray)

Same for this other paper (https://www.hindawi.com/journals/mpe/2013/419043/tab3/) where some better cost are found but with some constraint violation.

I perhaps miss something ?

8.4 Larger, more complex design optimization problem

My feeling up to know is that each problem is different and it is not so easy to generalize. It seems to me that a good understanding of the "shape" of the universe (even if it is not easy on high dimension) help the choice and the tuning of the algorithm.