A tool to compare Constrained Optimization problems with PSO algorithms

Rodolfo Saraceni
dept. name of organization (of Aff.)
Padova, Italy
rodolfo.saraceni@studenti.unipd.it

***Abstract*—The power and ease to use and configuration are the main features that have made the Particle Swarm Optimization method so common. Its simplicity of implementation has allowed the spread of an increasing number of algorithm with different techniques. Each of them present different performances but, quoting the No free lunch theorem [1], none is perfect and suitable for each type of problem. Thus a tool that allows you to compare the results of different algorithms on the same problem could be useful. This is the aim of this work: provide a simple application to evaluate the behavior of different type of PSO algorithms with our problem, using different techniques to solve it.**

***Index Terms*—particle swarm optimization, constrained optimization problem, compare tool**

I. Introduction

Optimization problems are very common questions in numerous applications like structural optimization, economics, engineering design and many others scientific fields.
A general optimization problem is defined like a function to minimize:

$$\min_x f(x) = \vec{x}^*$$

$$\vec{x} = (x_1, x_2, x_3, ..., x_n) \in \Re^n$$

The function $f(x)$ is called objective function and have dimension n, and each possible solution is represented by a vector $\vec{x}$ with same dimension. The one that best fitting the function is chosen as its minimum $\vec{x}^*$. These problems are defined unconstrained becouse they don't have some limitation in the computation for its minimum value.
In real problem different factors contribute in the resolution process and not all solutions found are acceptable. These limitations are mathematically represented by some equalities and/or inequalities:

$$h_{k1}(x) = 0, \qquad k_1 = 1, 2, ..., m_1$$

$$g_{k2}(x) \leqslant 0, \qquad k_2 = 1, 2, ..., m_2$$

with m_1 and m_2 respectively the number of equalities and inequalities constraints. Remember that an inequality $g(x) \geqslant 0$ could be converted in $g(x) \leqslant 0$ by multiplying by -1. Position that respect all the constraints is considered feasible, while, in the opposite the others are unfeasible.

In addition in real situation problems are definied in specific ranges where the parameters couldn't assume all possibile values but are limited in some intervals, so the function is bounded:

$$x_n^l \leqslant x_n \leqslant x_n^u$$

where the x_n^l represents che lower bound for the n-dimension while the x_n^u the upper one.

II. Generic PSO algorithm

In 1995 Eberhart and Kennedy [2] proposed an algorithm based on the natural behavior of cluster of elements: for example flocks of birds or schools of fish. The algorithm consists of a series of iterations, in each of which each element updates its position based on a set of current parameters and information based on previous iterations.

They consider that each particle tends to move towards its best situation considering two main factors: the past information gatered up to now and the needed to discover new better position. These aspects are matematicaly simbolized by ω and $pBest = (x_1, x_2, ..., x_n)$ called respectively inertial weight and personal best position. The first is a number to control the velocity of each particle: a large inertial weight help the global exploration, while a smaller one ease the exploitation of the best position. One solution often adobpted is to decrease its value during the algorithm process in a typically range from 0.9 to 0.1 [3]. The last is the best position reached until now by this particle.

This kind of algorithm also consider that each particles is part of a group and its movements are influenced by the nearby particles: to represent this actions they introduced the global best position $gBest = (x_1, x_2, ..., x_n)$ and other two parameters c_1 and c_2 called cognitive and social. Depending on the values attributed to these parameters a personal (higher c_1) or global (higher c_2) behavior of the particles is obtained. Their values could be constant (typically about 2) or could change during the iteration process in order to help a personal exploration in the first occurance in favor of a global exploitation in the last iterations
The constriction factor χ is introduced to ensure optimal trade-off between exploration and exploitation.

The new velocity of each particles, based on its previous velocity V_i^k and previous particle's position

$X_i^k = (x_1, x_2, ..., x_n)$, is defined by

$$V_i^{k+1} = \chi(\omega V_i^k + c_1 r1_i(pBest_i^k - X_i^k) + c_2 r2_i(gBest^k - X_i^k))$$

The two numbers $r1$ and $r2 \in [0,1]$ are random indipendetly generated and represents the randomness of the movements. The current iteration of the algorithm is indicated by k.
The new position reached with this velocity is computed with

$$X_i^{k+1} = X_i^k + V_i^{k+1} \qquad i = 1, 2, ..., D$$

III. Tool structures

The structure of this tool is very simply and linear. The first step of the process solution computing is to generate a initial random population to use to minimize the objective function. Then this population is processing by the chosen PSO algorithm. At the end the solutions is proposed to the user accompanied by graphs, also by a statistical treatment if the user computed multiple runs. The user could tuning each parameters of the PSO in order to adapt better the tool to own needed.

A. Generation of population

The initial population is the starting-point of the computation. The tool have different possibilities to generate them. Each one develop a population of an user-defined number of particles D, that represent the size of the population, with element's dimension equal to n. The possibilities implemented are:

- Non constrained random population: only the bounds are considered, so the result is a population uniformly distribute in the space defined by the boundary for each dimension. This is be useful for PSO that use a penalty function approach. It very fast because have a very low constriction.

- Constrained random population: it is the same of the previous method, but in this case each particle must verify each constraints (both equalities and inequalities). The time of computing is more, but it allow to be used in simpler algorithms like those use a True/False penalty approach).

- Opposite population: this is a possibility that user can apply or not. After generating a population, with one of the methods above, he can compute an opposite population [5] in order to try to improve the results obtained.

B. PSO algorithms

Three PSO algorithms are implemented, plus there is the possibility to use the best Python library about it. Each one have a different approach to solve the CO problems, but all of them could treat a problem defined as in the introduction.

- True/False penalty [6]: the procedure is very simple, it base its operation on a True or False verification of constraints. This cause strong difficult to handle equalities constraints. The main drawback is that the initial population must verify all the constraints before start to compute. Obviously only the elements with a True validation are consider feasible so a possible solution.
  Parameter ω is randomly generated each iteration for the entire swarm, $Vmax$ is set to the dynamic range of the particle $[10, -10]$. The steps are summarized below

```
For each particle {
    Do {
        Initialize particle
    } While particle is in the feasible space (i.e. it satisfies all the
    constraints)
}

Do {
    For each particle {
        Calculate fitness value
        If the fitness value is better than the best fitness value (pBest)
        in history AND the particle is in the feasible space, set current
        value as the new pBest
    }
    Choose the particle with the best fitness value of all the particles as
    the gBest
    For each particle {
        Calculate particle velocity according equation (a)
        Update particle position according equation (b)
    }
} While maximum iterations or minimum error criteria is not attained
```

Fig. 1. Main steps of first type of PSO algorithm

- Penalty function [7]: the approach of this method is to use a penalty value that is computed by a specific expression concerning the constraints violation. The value returned is add to the fitting value of each particle

$$F(x) = f(x) + h(k)H(x)$$

  where $f(x)$ is the original objective function of the CO problem, $h(k)$ is a dynamically modified penalty value, with k is the algorithm's current iteration number and $H(x)$ is a penalty factor, defined as

$$H(x) = \sum_{j=1}^{m} \theta(q_j(x)) q_j(x)^{\gamma(q_j(x))}$$

  where q_j is the numeric value of violation, m the number of constraints, and θ and γ two specific multi-stage assignment function.

- SAPSO 2011 [8]: This is the most complex algorithm of the tool. It use the reasoning behind the PSO, but with two main differences respect the generic expression defined in the PSO section. The first is about updating of the position that in not only based on the velocity but have a random component

$$V_i^{k+1} = \omega_i V_i^k + X r_i^k - X_i^k \qquad i = 1, 2, ..., D$$

where V and X are always particle's the velocity and the position, while Xr_i is a point randomly drawn in a hypersphere centered in the isobarycenter of the swarm G_i^k and radius equal to $\|G_i^k - X_i^k\|$. The other difference is that the parameters not have fixed values, but they adapt along the iteration process depending to the movement of the swarm, helping the exploitation or the exploration as needed.
Also employ a similar penalty function of the last algorithm, but use the violation value to classified the solution in feasible or not, choosing the $pBest$ and $gBest$ by a feasibility based rule.

- PYSWARM [9]: is the most complete library in Python about PSO. It allow to compute a CO problem tuning its parameters: population size D, number of iteration, inertial weight ω, cognitive c_1 and social c_2 coefficients and also we can provide a minimum step size of $gBest$ and minimum change of $gBest$ before terminates the process. It require, obviously the objective function, and its constraints and boundaries.
  To be able to use Pyswarm it was necessary to readjust the inserted functions to be handled by the library that uses constraints of inequality greater than or equal to zero. This method could be helpful to have a comparison with an external and already tested tool.

C. Graphs and results

The result is provided in both text and graphical way. For each run a progression bar is displayed the time employed to generate the population and to run the PSO respectively. Also the gBest's value and its position in the space is given. If they were made multiple runs a statistical treatment of all results is provided: returning the standard deviation, the mean, worst and best solution, and, if the best minimum known is provided, it return the absolute error too.

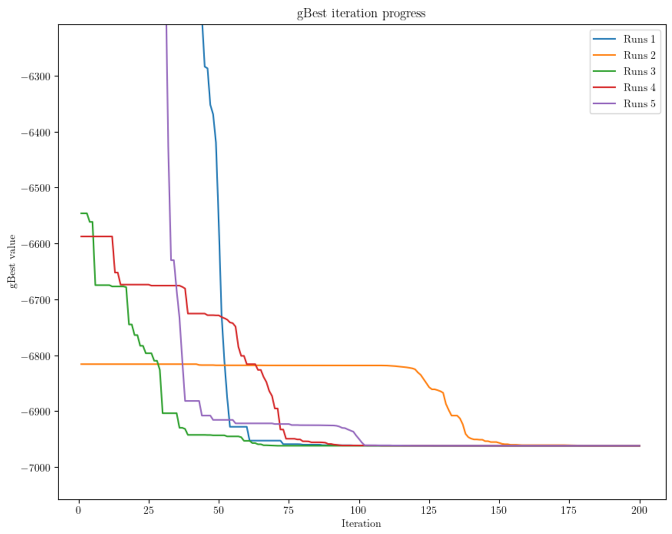

Fig. 2. Example of a fitting value of gBest during the iteration progress.

The graphs displayed are two: Figure 2 is a line plot of the progression of the gBest's value along the iteration process, while Figure 3 is an animation of the movement of entire swarm of particles during each iteration, very useful to analyze and understand the operation of the PSO algorithm, especially in two dimensional cases.

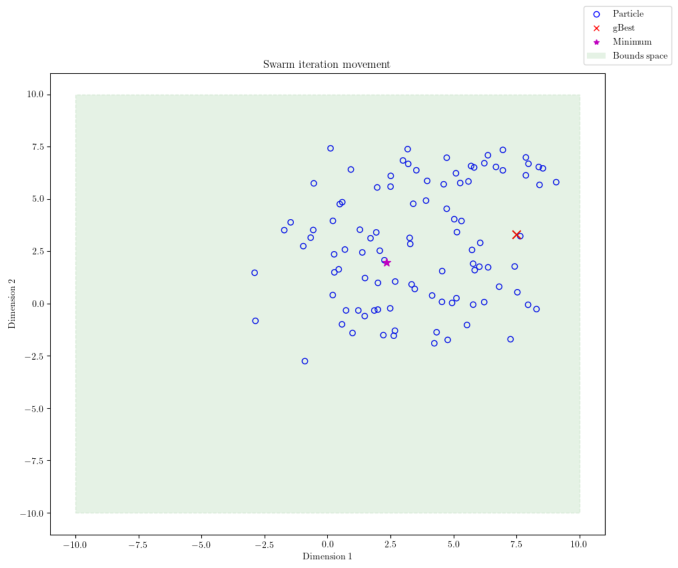

Fig. 3. Snapshot of the animated graph of swarm movements during the iteration progress.

D. Benchmarck function

User can insert his problem to minimize in the tool simply defining a python class in the functions file that is retained separate from the source code. He can provide a lot of parameters, in order: name of the function, dimension n, objective function, best minimum already found, position of best minimum in the n space, equalities constraints (without =0), inequalities constraints (without ¡= 0), lower and upper bound for each dimensions. But only the objective function and the constraints are essential.

The tool provide a predefined set of benchmark functions with respectively minimums and their position for evaluate the different PSO algorithms [6]–[8].

```
g1_6 = Bench_func("g1_6",
          2,
          "(x1-10)**3 + (x2-20)**3",
          -6961.81381,
          [14.095,0.84296],
          [],
          ["100 - (x1-5)**2 - (x2-5)**2",
          "(x1-6)**2 + (x2-5)**2 - 82.81"],
          [[13,100],
          [0,100]])
```

Listing 1. An example of a simple benchmark function

IV. Conclusion and further work

PSO is a very useful instrument for solving a very large amount of problems. Its simplicity of implementation led to writing a large number of algorithm which use different techniques. This wide choice is a very good opportunity but often is not so easily available because the most of them are only specific research or study and the codes are non directly accessible. Enable a quick use both for newbie and professionals figures is the main goal of this tool.

Concerning the further implementations must consider that to preserve the aim of this tool a continuous updating of the algorithms available and the concerning processes connected to them is necessary. In addition to this, it would be interesting to develop a faster and more equally distributed generation of the initial population in the space delimited by bounds. This could be possible using the Monte Carlo method or even better the Sobol sequences [4].
Also might be interesting develop a local version of the algorithm, where each particle non consider the entire swarm for its social behavior but only its neighbors. The user could insert a radius , in terms of Euclidean distance, within which consider the other particles neighbors.

In order to simplify the usability by the final user a GUI could be very helpful. It would speed-up the setting phase and the insertion of the equation representative the problem to minimize, in fact the user will no longer have to write lines of code but will simply have to enter the values of various parameters in numerical format in an intuitive environment.

In a spirit of sharing, referring back to the principles of free software and hoping that it may be of help to someone else, all the code written is available on Github [10].

References

[1] D.H. Wolpert, W.G. Macready, ”No Free Lunch Theorems for Optimization”, IEEE Transactions on Evolutionary Computation 1, 67, 1997.

[2] J. Kennedy, R. Eberhart, ”Particle swarm optimization” Proceedings of ICNN’95 - International Conference on Neural Networks, IEEE, vol. A247, 1995.

[3] S. Sengupta, S.Basak, R.A. Peters, ”Particle Swarm Optimization: A Survey of Historical and Recent Developments with Hybridization Perspectives”, Dep. of Electrical Engineering and Computer Science, Vanderblit University, October 2018.

[4] W.H., Press, W.T. Vetterling, S.A. Teukolsky, B.P. Flannery, ”Numerical Recipes in Fortran 77”, Cambridge University Press, Cambridge, 1992.

[5] H. Jabeen, Z. Jalil, A. Rauf Baig, ”Opposition Based Initialization in Particle Swarm Optimization (O-PSO)”, GECCO, Montréal Québec, Canada, July 2009.

[6] X. Hu, R. Eberhart, ”Solving Constrained Nonlinear Optimization Problems with Particle Swarm Optimization”, USA.

[7] K.E. Parsopoulos, M.N. Vrahatis, ”Particle Swarm Optimization Method for Constrained Optimization Problems”, Dep. of Mathematics, University of Patras Artificial Intelligence (UPAIRC), Greece.

[8] B. Tang, Z. Zhu, J. Luo, ”A Framework for Constrained Optimization Problems Based on a Modified Particle Swarm Optimization”, László T. Kóczy, June 2016.

[9] Tisimst, ”pythonhosted.org/pyswarm/”, 2014

[10] R. Saraceni, ”github.com/rodosara/PSO-tool”, 2019