**Group Members:**

Syeda Areesha Najam (sn05985)

Shalin Amir Ali (sa06132)

Sana Fatima (sf06199)

**Project Title: Particle Swarm Optimization (PSO)**

**Abstract:**

This project will introduce the particle swarm optimization (PSO) algorithm as a stochastic algorithm used for solving optimization problems. So as to officially introduce the scientific detailing of PSO algorithm, the classical inertial version of PSO will be used, meanwhile PSO variants will be summarized. Before looking into implementation, this project will introduce important concepts and functions of PSO. Based on this knowledge, mathematical model of PSO algorithm will be formulated. Other than this, the theoretical analysis and experimental analysis will be discussed. Two study cases of optimizing functions, one of ‘Sphere’ and another is ‘Rastrigrn function’, are provided for implementation of PSO algorithm. This is to show how handful and versatile it is to work with PSO.

**Introduction:**

Particle swarm optimization (PSO) is one of the rare tools which is amusingly easy to code and implement to produce bizarrely good results. This algorithm was proposed by Eberhart and Kennedy in 1995. It is a population based stochastic, something randomly determined, algorithm which is used to solve a numerical optimization problems like predicting score of a football team using a math equation. According to this algorithm, basically, goal is to minimize error terms (difference between actual answer and predicted answer). Such goal categorizes PSO as metaheuristic approach which means that a higher level procedure is used to find optimal solution for any optimization problem with imperfect data or limited computation capacity.

PSO, a unique computational method, is inspired from social behaviors of nature. By social behavior we mean, the collective behaviors of simple individuals interacting with their environment and each other like social foraging behaviors of birds’ flocking, schooling of fishes etc. Hence, it is also classified as swarm intelligence algorithm like other bacterial foraging algorithm, ant colony algorithm etc.

**Important concepts and functions of PSO:**

Particle Swarm Optimization is a population based method. The word “swarm” refers to the collection of particles which acts as flock of birds which are searching for food (a biological phenomenon). In nature, there is some boundary till where the birds can reach while they are in searching. Similarly, in this algorithm, the boundary is named as a terminal point. Thus, the particle can approach a saddle point in which the slope (in our case, its velocity) will be zero. The swarm of particles helps this method to run efficiently as none of the particles get stuck in local minima. In fact they follow the particle which is closest to the terminal point. The mathematical formulation of this algorithm to find the global minima is described as follows:

1. The algorithm initializes with a group of random particles, each having different position and moving with different velocity.
2. After every iteration, each particle updates its best position which it has attained till then, known as the “personal best” or ‘pbest” of that particle which helps it to determine its current velocity to some extent.
3. This function also keeps a track on the best position attained by the group, known as “global best” or “gbest”, which affects the velocities of the entire swarm at a definite rate.

When the swarm of particles reach at the terminal point, this algorithm stops functioning and returns the optimum solution.

**Mathematical model of PSO:**

For every particle two vectors are considered, the velocity vector and the position vector. The position vector shows the position of a particle in certain landscape and the other, velocity vector shows the intensity and direction of movement of that particle. There are two equations mentioned below, which make up back bone of PSO. Note that the “k” in equations denotes the current iteration, therefore “k+1″ implies the next iteration.

Position of individual particles updated as follows:

With the velocity calculated as follows:

Where,

|  |  |
| --- | --- |
|  | Particle position |
|  | Particle velocity |
|  | Best individual particle position |
|  | Best swarm position |
|  | Constant inertial weight |
|  | Cognitive and social parameters |
|  | Random numbers between 0 and 1 |

And,

* is the inertial component
* Is the cognitive component.
* Is the cognitive component.

The first equation for position of particle tells that the next day’s position of individual particle is calculated by summing today’s position of particle and its velocity for the next day. Then, in order to calculate particle’s velocity for next day, the inertial component, the cognitive component and the social component are summed up in second equation. In this equation,  gives distance to the personal best and gives distance to the global best. The cognitive component helps particles in exploring the search space and the social component helps particles in exploiting the search space. Note that velocity in current day is used to calculate the velocity for next day and this is how it helps in deducing the position for next day.

**Impact of variants:**

Analysis of inertia weight selection:

The inertial parameter of PSO tunes the exploitation and exploration of particles and also shows the influence of previous velocity on the current velocity. If w = 0, the velocity of particle depends on its personal best and global best. On the other hand, if w ≠ 0, then the particle has the tendency of exploring new space. The larger the inertia weight, the greater the velocity and exploration rate will be. Similarly the smaller the inertial weight, smaller will be the velocity and particle will tend to perform local exploitation.

Analysis of selection of cognitive and social parameters:

These parameters are also known as accelerating constants and are responsible to represent stochastic acceleration of particle towards its personal best point and the global best point. From the equation above we can infer that if c1 = c2 = 0, then particle will fly with constant velocity till the border. This makes difficult for particle to approach optimum point because then the particle only have ability to search in definite local space.

Similarly if only c1 = 0, particle losses its cognitive ability and depends on its social component, decision made by whole swarm so far. This increases convergent rate of particles and exploitation is done at highest level. In contrast to this, the particle loses its social component when c2 = 0. This means that every particle search in its local space and do not exchange information with one another. In this case, the exploitation is done at its lowest level and exploration is done at its highest level.

With all of this, we can conclude that, higher value of the cognitive parameter results in excessive exploitation in local search space of individuals. On the other hand, relatively higher value of social parameter provides tendency to particle to rush prematurely towards the optimum point.

**Time Complexity of PSO:**

The worst case time complexity of the algorithm is O (n2). This algorithm starts with initializing the particles population. The “do while” loop runs until the particles find their optimum solution (O (n)) or they reach the maximum iteration point (it is a numerical value, hence it doesn’t affect the complexity). The first nested “for” loop runs for ‘n’ (number of particles) times inside which the function calculates the fitness value of each particle and updates the personal best position if needed. Therefore, all the work done inside the first “for” loop has a complexity of O (1). The algorithm then defines/updates the global best position (O (1)). Moving on to the second “for” loop which runs for ‘n’ times. Inside this loop, the velocity and position is determined (O (1)).

Hence, the total time complexity of this algorithm is O (n (n+n)) which is equivalent to O (n2).

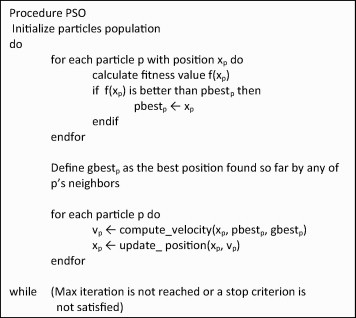


Fig 1. Pseudo code for basic PSO.

**Implementation and Experimental Analysis of PSO:**

For our implementation we took reference of the PSO code of Robert Green from his GitHub repository- rgreen13. Various modifications were made to make our code efficient while we implemented the referenced PSO code on our own of which the major change was that our objective function was different which we aimed to optimize. In order to validate our implementation of PSO algorithm used PySwarms library. PySwarms is a research toolkit for Particle Swarm Optimization (PSO) that provides a set of class primitives useful for solving continuous and combinatorial optimization problems (James Miranda). We compared our PSO implementation with PySwarms on the basis of running time of both implementations with the given set of data frames to understand the effect of size problem on PSO. Since our objective was to find the global best of the Sphere function and Rastrigin function, we used **pyswarms.single.global\_best module** on both functions individually. In our experiment we kept c1 = 0.5, c2 = 0.3, and w = 0.9 constant and evaluated performance of our PSO on three test cases:

* Varying number of iterations (Num\_Iterations) in the range [100, 3000] with 100 step interval and keeping Num\_Particles = 20 while Num\_dimensions = 2
* Varying number of particles (Num\_Particles) in range [10, 500] with 10 step interval while keeping Num\_Iterations = 50 and Num\_dimensions = 2
* Varying number of dimensions (Num\_dimensions) in the range [2, 25] with 1 step interval while keeping Num\_Particles = 20 and Num\_Iterations = 50.

We optimized Sphere function and Rastrigin function using the same test cases to find the global minimum within the bounds of search space i.e. in given number of dimensions. Also the velocity was kept within the bounds i.e. .

Comparison of Runtime of PSO and PySwarms:

1. PSO sphere and PySwarms:

* Varying number of dimensions

Fig 2. . Comparison of runtime (µs) of PySwarms and our PSO implementation with varying number of dimensions.

Comparing our runtime of PSO\_sphere with PySwarms’ runtime, it is noticeable that when the number of dimensions were less than 13, our PSO code proved to be more efficient whereas for greater dimensions, dimensions greater than 13, the runtime of PySwarms proved to be more efficient. It is noticeable, looking at the graph that there exists some points for which the runtime of both (PSO and PySwarms) is same e.g. 13. We can see when the number of dimensions is 25, our PSO\_sphere has a runtime less than PySwarms (a difference of 8,730 µs). The results conclude that PSO\_sphere is less efficient at handling dimensions between 13 and 24 as compared to PySwarms but at 25 dimensions, its runtime is much lesser compared to that of PySwarms.

* Varying number of iterations:

Fig 3. . Comparison of runtime (µs) of PySwarms and our PSO implementation with varying number of iterations.

Fig 3 shows that as the number of iterations increased, the runtime of both PSO\_sphere and PySwarms also increased. We can see that as the number of iterations is increased, there is a drastic change in the runtime of PySwarms as compared to our implemented code of PSO. While contrasting both graph lines, it is noticeable that our PSO sphere is more efficient as compared to PySwarms. Especially for 2300 iterations, there is a difference of 608,257 µs (greatest difference). Hence it can be concluded that the performance of PSO\_sphere was more efficient than PySwarms while handling greater number of iterations.

* Varying number of particles:

Fig 4. . Comparison of runtime (µs) of PySwarms and our PSO implementation with varying number of particles.

Fig 4 shows that as the number of particles increased the runtime of PySwarms implementation remained the same with some fluctuations. Differentiating between PSO\_sphere’s runtime and PySwarms’ runtime, Fig 4 shows that runtime of PySwarms remained between the ranges of 4000 µs to 7000 µs whereas our PSO implementation proved to be efficient for 130 or less number of particles. It can be concluded that our implementation of PSO proves to be less efficient (if we are running the code for 130 or more particles) as compared to PySwarms as its runtime is almost the same for increasing number of particles.

1. PSO\_rastrgin and PySwarms:

* Varying number of iterations:

Fig 5. Comparison of runtime (µs) of PySwarms and our PSO implementation with varying number of iterations.

While experimenting, the number of iterations is increased from 100 to 3000 with an interval of 100 steps. In both implementations, it can be seen from graph that growth rate of both curves is steady with some fluctuations. Also similar trend of peaks is observable for both curves in Fig 5. This ensures resemblance of our implementation with PySwarms. In addition to this, from graph we can observe that with increasing number of iterations, the runtime of PySwarms is increasing rapidly with respect to our implementation. The use of word rapid indicates steeper slope of curve. On the other hand, we see that runtime of our implementation shows steady growth rate with much lesser rapidness and lesser steep curve than PySwarms. At most, with 3000 iterations, the difference of 282000 µs for runtime of both was recorded. From all these observations it can be concluded that our implementation is more optimized and efficient than PySwarms implementation.

* Varying number of Particles:

Fig 6. . Comparison of runtime (µs) of PySwarms and our PSO implementation with varying number of particles.

In our test values, the number of particles is increased from 10 to 500 with an interval of 10 steps. From above mentioned graph, it is clearly stated that runtime of PySwarms implementation keeps fluctuating roughly from 25000 µs to 44000 µs. On the other side, a steeper curve for runtime of our implementation shows steady increase in runtime with increasing size of swarm. However, the lesser steep curve of PySwarms implementation tells that it is more efficient in handling increased size of swarms than our PSO implementation. In addition, the trend of peaks in both curves show pattern of fluctuating runtime. The runtime of our PSO implementation is consistent and have quite rounder peaks, indicating regular change in values in some interval. Whereas, the runtime of PySwarms kept fluctuating throughout, shows sharp peaks in curve. Collectively, we can say that with respect to time consumption, PySwarms implementation is more efficient and with respect to stability our PSO implementation is more better..

* Varying number of dimensions:

Fig 7. Comparison of runtimes (µs) between PySwarms and our PSO implementation with varying number of dimensions

While validating the impact of increasing number of dimensions, the growth of PySwarms and PSO\_rastrigin was roughly linear. The efficiency of handling size problem with respect to number of dimensions of PySwarms and PSO\_rastrigin were same. Thus, it can be concluded that our PSO implementation is efficient while execution and is similar with the published PySwarms library.

**Applications of PSO:**

PSO can work effectively for all those optimization problems which cannot be mathematically modeled. Applications of PSO are found in variety of fields ranging from engineering design, process optimization, to service oriented applications in healthcare and bioinformatics. Some of them are listed below.

* The Analysis of Basketball Free Throw Trajectory using PSO Algorithm:

PSO algorithm can be used to track the ball while playing basketball. The research determined that there is a significant relation between the height and shape of free throw trajectory. This method can also be used while training the players in order to improve their performance of free throw.

* Heating System Planning Problem:

The objective of HSP is to minimize the total coast of heating system production. The proposed integral mathematical model of PSO minimizes the cost of heating system as the objective for a given life cycle time. It is believed that if this approach is applied correctly and in combination with other elements, it can become a powerful and effective optimization tool for HSP problem.

**Conclusion:**

From the experimental analysis, we can conclude that our PSO implementation was efficient in handling varying number of dimensions and number of iterations which shows that our PSO implementation can further be used and improved for applying it to optimize other set of functions. PSO code can further be mapped on other applications and to make the basic PSO implementation more robust and advanced.

**References:**

* [**https://www.researchgate.net/figure/The-pseudocode-of-the-PSO-algorithm\_fig8\_274460300**](https://www.researchgate.net/figure/The-pseudocode-of-the-PSO-algorithm_fig8_274460300)
* [**https://github.com/rgreen13/PSO-Python**](https://github.com/rgreen13/PSO-Python)
* [**https://www.hindawi.com/journals/tswj/2014/973093/fig1**](https://www.hindawi.com/journals/tswj/2014/973093/fig1/)
* [**https://pt.slideshare.net/MohamedTalaat9/digital-image-forgery?next\_slideshow=1**](https://pt.slideshare.net/MohamedTalaat9/digital-image-forgery?next_slideshow=1)
* [**https://slideplayer.com/slide/12229714**](https://slideplayer.com/slide/12229714/)
* [**https://nathanrooy.github.io/posts/2016-08-17/simple-particle-swarm-optimization-with-python**](https://nathanrooy.github.io/posts/2016-08-17/simple-particle-swarm-optimization-with-python/)
* [**http://benchmarkfcns.xyz/benchmarkfcns/spherefcn.html**](http://benchmarkfcns.xyz/benchmarkfcns/spherefcn.html)