An In-Depth Analysis on Particle Swarm Optimization

Rohan Chhaya

ETH Zürich, Spring 2023

Contents

All figures not cited are created by the author.

The source code can be found at this Github repository

1 Algorithm Background

1.1 Motivation and Comparison to Other Algorithms

Particle Swarm Optimization, also referred to as "PSO", is an algorithm that tries to mimic biological patterns akin to flocks of birds or a swarm of mosquitos. Using a single walk for optimization may not be effective because it may be ignoring large parts of the function space, and multi-point algorithms like Nelder-Mead often only update a subset of points or even a single point every iteration. A paradigm shift toward population-based systems where each particle updates itself based on social interactions seems beneficial. This also plays into the exploration-optimization trade-off of many modern algorithms, since a swarm of particles may do a better job of exploring the objective function space.

While "swarm intelligence" is questionable and difficult to implement, it is surprisingly easy to simulate a simplified version with a group of particles.

Each particle only knows its current position and velocity, the "best" cost it achieved and its location, and the "best" cost that its neighbors have achieved with its location. Complex group dynamics are ignored and this primitive memory and physics prove to be an effective, simple solution. There are lots of variations to add more swarm dynamics into the mix, such as adding mass and acceleration states to each particle

1.2 Mathematical Details

In its core form, the formula for PSO is as follows:

$$\mathbf{p}_i(t+1) = \mathbf{p}_i(t) + \mathbf{v}_i(t+1)$$

$$\mathbf{v}_i(t+1) = w \cdot \mathbf{v}_i(t) + c_1 \cdot r_1 \cdot [\mathbf{p}_{\text{personal best}}(t)) - \mathbf{p}_i(t)] + c_2 \cdot r_2 \cdot [\mathbf{p}_{\text{neighbor best}}(t)) - \mathbf{p}_i(t)]$$

Each particle moves one step in the direction of its current velocity, and the velocity is updated to be a combination of the current velocity (with inertial parameter w), a pull toward its personal best (with cognitive parameter c_1 and random number r_1), and a pull toward the best of its neighbors (with social parameter c_2 and random number r_2). While this algorithm appears quite simple and is bio-inspired, the use of hyperparameters and random numbers together creates complexities with lots of tuning screws and variations possible.

Each particle feels forces pulling it in different directions, with a larger distance (larger $\mathbf{p}_{\text{best}}(t)) - \mathbf{p}_i(t)$) corresponding to a larger force felt. The idea is that as the algorithm progresses, one of the particles will stumble onto a minima and that will influence the swarm to move toward it, but with lots of noise and randomness to still explore the objective function space.

There has been lots of research to prove the particles in the algorithm converge and whether they converge to a local minimum. While it was shown that simplified models of PSO do show reliable particle convergence for a variety of implementations, convergence to a local minimum is not as robust [Cleghorn and Engelbrecht, 2017]. There are lots of empirical evaluations and modified-PSOs that result in convergence to a minimum, like GCPSO, where the "best" particle is given a different mechanism to update its position and velocity [van den Bergh and Engelbrecht, 2010]. However, the theory to show convergence to a minimum for the general-form PSO is lacking. Nevertheless, because of its simplicity, lack of gradient calculations, and empirical effectiveness, it remains an extremely popular optimization algorithm.

2 Library Implementation and Hyperparameter Identification

2.1 PySwarms Information

From online research and speaking with machine learning engineers, the consensus software used, `PySwarms`, is very effective. This library was used in the analysis that follows, but implementation from scratch was also done.

This library, first proposed and written by [Miranda, 2018], has multiple implementations: General PSO, Global-Best PSO, Local Best PSO, and Discrete PSO.

The first 3 algorithms are meant for continuous problems, while the Discrete PSO implementation is meant to be used only for discrete situations. The General PSO implementation allows the user to specify any `Topology` object to determine how the swarm communicates

- Star: all connected together
- Ring: particles connected to k nearest neighbors
- Von Neumann: A Von Neumann topology that is somewhat similar to a cube-like connection
- Pyramid: particles are connected via simplexes
- Random: particles are connected to k random particles

The Global-Best PSO (GB) is an implementation with a star topology, while the Local-Best (LB) PSO uses a ring topology. It is generally accepted that the LB optimizer takes more time to converge, but has a better exploration of the objective function space compared to the GB optimizer. This is because each particle doesn't compare itself to the swarm optimum, but rather, the optimum of its nearest neighbors. However, note that [Engelbrecht, 2013] provided a meta-analysis and benchmarking study and showed similar performance between the two algorithms when it comes to convergence, exploration, and solution accuracy. So, this analysis will be using both for different purposes.

2.2 Hyperparameter Selection

The hyperparameters that can be passed into all the variations are as follows:

- c_1: The cognitive parameter, which controls the pull of the particle towards its current best
- c_2: The social parameter, which controls the pull of the particle toward the best of its neighbors
- w: The inertial parameter.
- n_particles, iters: For both of these parameters, performance generally improves at higher values at the cost of higher computation. Therefore, one generally uses the highest value possible given the technological or visualization constraints

Additionally, in the discrete, LB, and general form optimizer, the following additional hyperparameters are needed based on the topology

- k: The number of neighbors to be considered (needed if using the general optimizer with a Ring, Von Neumann, or Random topology
- p: the Minkowski p-norm to use, where 1 is L1, or Manhattan/taxicab distance and 2 is L2, or Euclidean, distance

3 Application 1: Hyperparameter exploration with Rosenbrock function

3.1 Rosenbrock Function

The Rosenbrock function is a popular test function for optimization algorithms. To visualize the algorithm, the objective function was used in two dimensions, where cost = $f(x, y)$ so the minima is located at (1,1) and is 0.

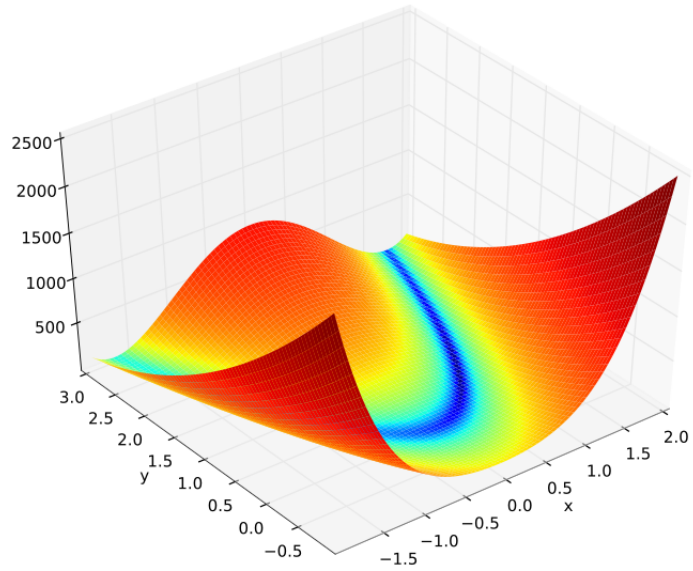


Figure 1: The Rosenbrock "banana" function in 3D space: $f(x, y) = (1 - x)^2 + 100 \cdot (y - x^2)^2$. Figure from [Saha, 2014]

This PySwams optimizer is also quite powerful, and setting the number of iterations or particle count to a very high number almost always finds the minimum of this test function regardless of hyperparameters. Thus, the algorithm implementation will use 100 iterations and 10 particles unless otherwise specified.

3.2 Hyperparameter Optimization with RandomSearch

First, this algorithm must be tuned to use the optimal hyperparameters. However, given the probabilistic and stochastic nature of this algorithm, there is not an "optimal" set of hyperparameters that guarantees optimal performance. So, a `RandomSearch` was run over the parameters of c_1, c_2, and w for the `GlobalBest` optimizer. The parameters c_1 and c_2 were varied between 0 and 1 (following best normalization practices) and w between 0 and 3 with 100 iterations each time to find the optimal score and parameters.

To account for the stochastic nature of the algorithm, this process was repeated 100 times and any results with optimal cost > 0.001 were removed.

First, let us look at the distribution of optimal c_1 (cognitive), c_2 (social), and w (inertial) over the 100 trials.

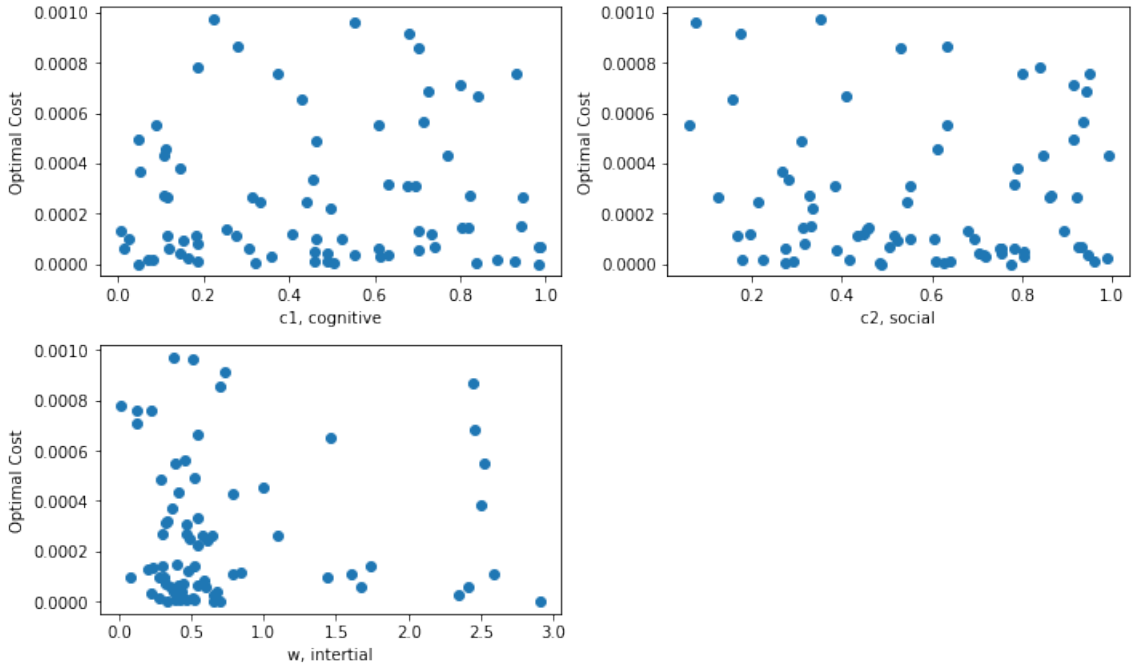


Figure 2: Plotting 100 iterations of hyperparameter optimizations gives high variability

There is a lot of randomness in the c_1 and c_2 variables, while there is a clear focus towards a 0.25-0.75 range for the w variable. If the plots are limited to only using the top 10 performing parameter tuples, a clearer picture may be present.

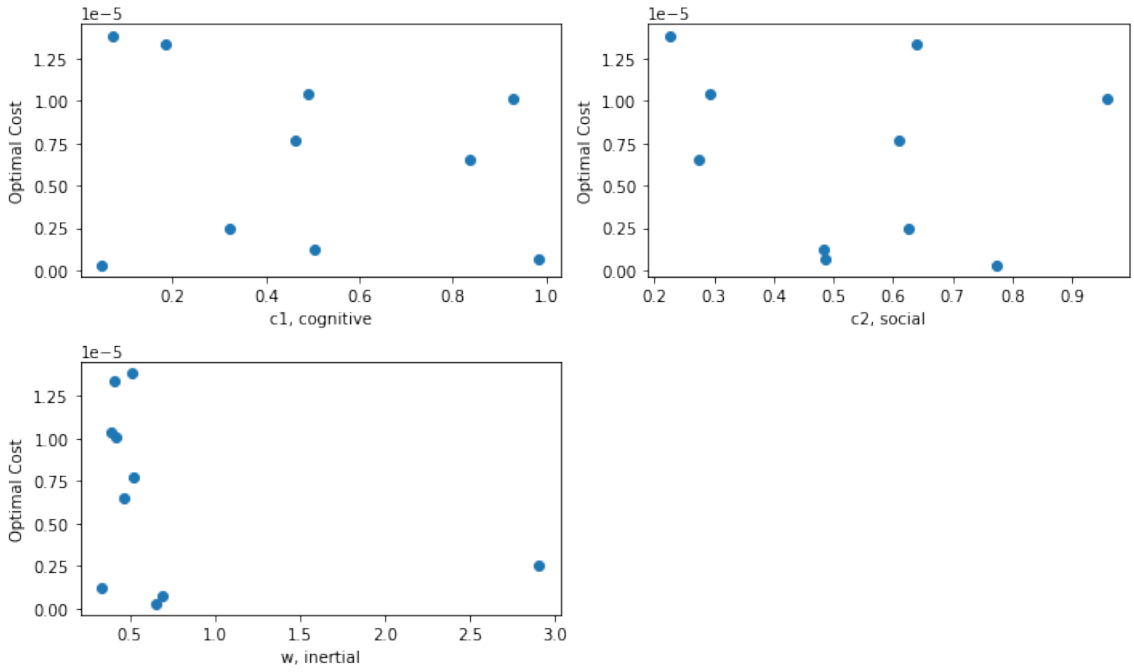


Figure 3: The scatterplot of the 10 highest performing iterations

Here, there is a bit more clarity in the w parameter towards $0.4 < w < 0.55$, but there is still a lot of randomness present in the c_1 and c_2 parameters. While c_2 tends to have the best results between

0.45 and 0.7, there is too much noise to consider this a legitimate trend, even with several layers of averaging and repetition. This indicates the high level of randomness and stochastic processes present in the algorithm that ensures the algorithm is not deterministic for a given hyperparameter set.

3.3 Hyperparameter Variation and Path Visualization

The model was run for various hyperparameter sets to observe the particle path and loss function each time and how each hyperparameter affects it. Below are some of the iterations, this time using `LocalPSO` to see the effect of k, the number of neighbors considered in the social calculation, and p, the norm used. While animations cannot be put into this report, the path of each particle was traced for visualizations. Note each particle was randomly initialized to a location between (-2, -2) and (2, 2)

As a default baseline, $c_1 = 0.5$, $c_2 = 0.5, w = 0.5, k = 10, p = 1$. This is based on the previous analysis where 0.5 was a suitable value for w, c_1, and c_2 were just chosen at the median of 0.5 due to their variability, k was chosen to be 10 to see the effects of the other variables more clearly, and p was arbitrarily chosen. The blue dots are the initial particle locations, the green dots are the final particle locations (may not be visible), and the red lines are the particle paths.

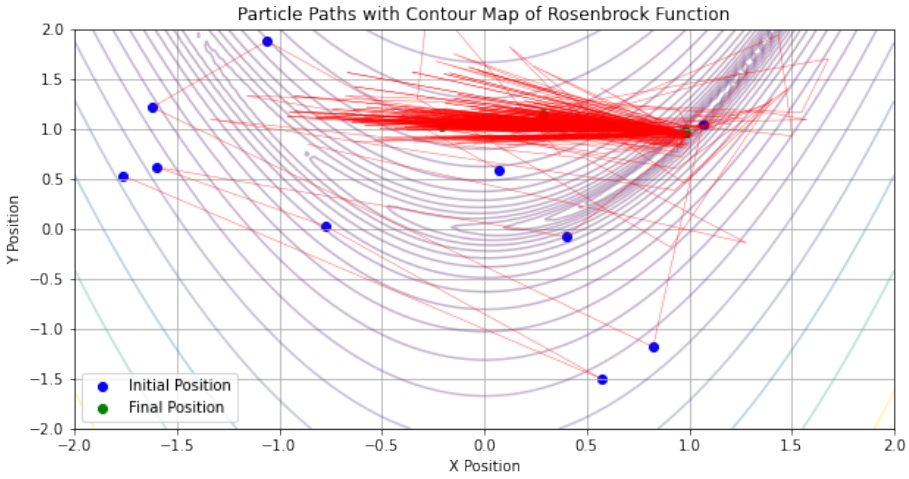
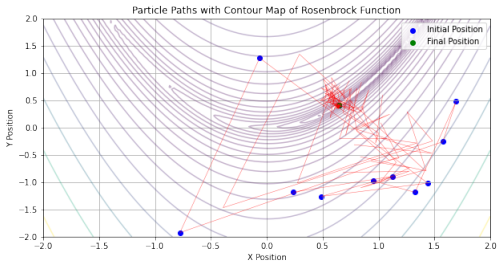
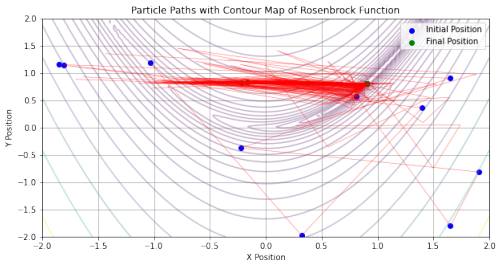
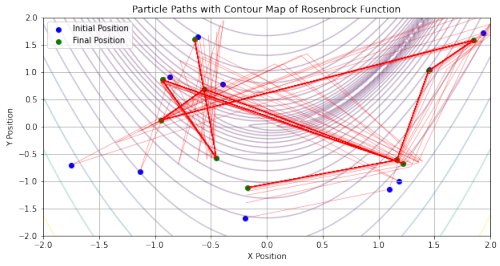
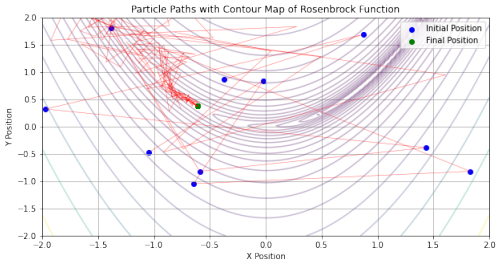
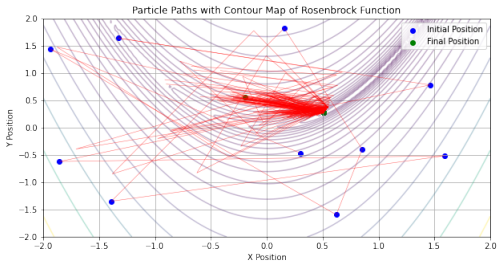
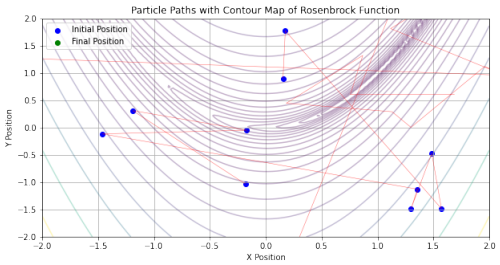
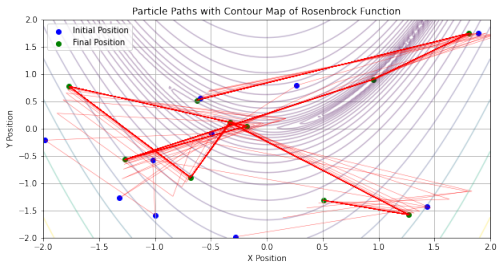
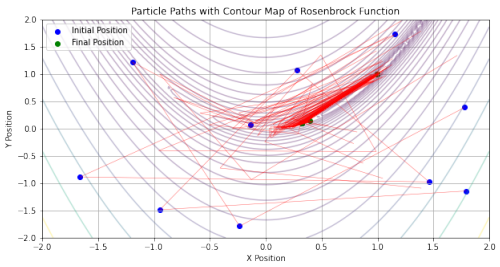
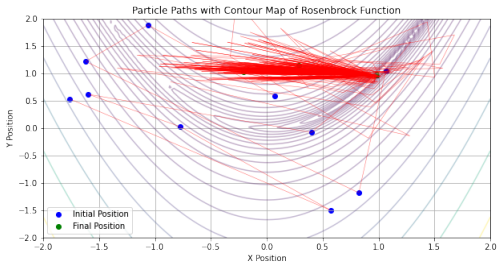
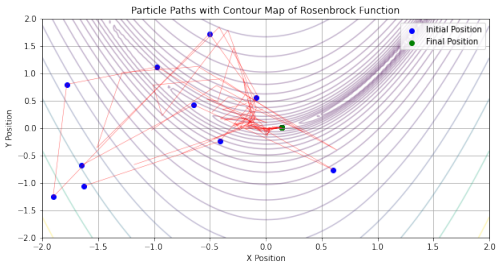


Figure 4: The resulting cost was 0.00025 with the position (0.984, 0.969). Note the optimal of 0 at (1, 1)

These variables are kept constant and changed one at a time to see the effect of each parameter individually. The extreme values were visualized to see the extent of the impact.

Table 1: Particle paths with various hyperparameter regimes

Parameter	**Value 1**	**Value 2**
c_1, cognitive	 $c_1 = 0$, best cost = 0.124 at (0.65, 0.42)	 $c_1 = 1$, best cost = 0.0101 at (0.90, 0.81)
c_2, social	 $c_2 = 0$, cost = 3.754 at (-0.93, 0.86)	 $c_2 = 1$, cost = 2.604 at (-0.61, 0.38)
w, inertial	 $w = 0$, cost = 0.257 at (0.51, 0.27)	 $w = 2$, cost = 0.203 at (0.76, 0.61)
k, # of neighbors	 k = 1, cost = 0.00230 at (0.95, 0.91)	 k = 5, cost = 1e-6 at (0.999, 0.998)
p, Minkowski p-norm	 Equivalent to baseline: p=1, cost = 0.00025 at (0.98, 0.97	 p = 2, cost = 0.751 at (0.14, 0.01)

Note to take the specifics of each cost/location with a grain of salt because they are just 1 iteration of a stochastic algorithm. There are, however, some patterns that emerge from the plots:

- A low c_1 parameter means the particles follow the swarm tightly. There are very few red lines except at the swarm optimal, where there is a high concentration. Conversely, a high c_1 parameter shows a more spread-out particle path profile and a concentration at multiple points in the valley of the function.
- A low c_2 parameter means the particles tend to follow their own paths toward their optimal more often, as shown by the 10 dense red lines for each particle. A high c_2 parameter shows the particles zeroing in on the swarm optimal quite early.
  - Note the high objective function cost values of these parameter choices. When $c_2 = 0$, the social behavior of the swarm is killed and the chance of finding the optimal is much more dependent on particle initial states. When $c_2 = 1$, the social behavior dominates and there is much less exploration of the objective function space

- A low w tends to pull the swarm to the swarm optimum faster, while a high w causes the particles to fly off into space. It is generally not advised to have a $w \geq 1$ for this reason, especially if the objective function has a strictly constrained domain.
- Changing k has a similar effect as the social parameter c_2: A very low k means fewer neighbors to be influenced by and thus more focus on individual behavior and movement. At higher k, there is much more social behavior and pull toward the swarm's optimal
- When $p = 1$, the particle paths seem to be more linear and gridline. When $p = 2$, there is more diagonal movement in the function space, reflecting the difference between Manhattan $L1$ distance and Euclidean $L2$ distance

Overall, note the high variability and dependence on initial conditions. If a point is randomly initialized close to the global minimum, there is a much higher chance to find it. However, it may be quite difficult if none of the points are close to it. Due to the randomness of the algorithm due to random numbers $r1, r2$, sometimes the behavior is unpredictable (e.g. When $c_2 = 0$, there is a particle initialized close to the optimal but does not get close at all).

4 Application 2: Boundedness and Resistance to Local Minima with the Hölder Table Function

4.1 Hölder Table Function

The Rosenbrock function is a great optimization test function due to its narrow, steep valley that the minimum occurs in. Another major test of optimization functions is not getting trapped in local minima on the search towards the global minima. The Hölder Table function has minima of -19.2085 at $(\pm 8.05502, \pm 9.66459)$ and has many local minima near the global minima. Note the domain of the function is on $[-10, 10]$ for x, y, which is different than the unbounded Rosenbrock function

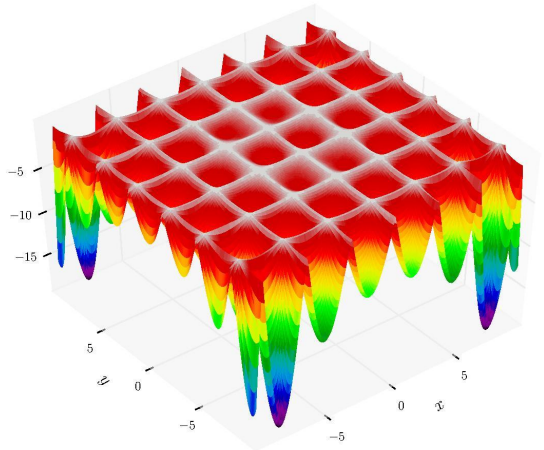


Figure 5: The Hölder table function, $f(x, y) = -\left|\sin(x)\cos(y)\exp\left(\left|1 - \frac{\sqrt{x^2+y^2}}{\pi}\right|\right)\right|$
Figure from [Al-Roomi, 2015]

4.2 Hyperparameter Variation and Path Visualization

Since the analysis on hyperparameter optimization using `RandomSearch` showed us that the optimal parameters are very noisy, this analysis will skip that step. Instead, the same baseline of $c_1 = 0.5$, $c_2 = 0.5, w = 0.5, k = 10, p = 1$ was used (see below) and each hyperparameter will be varied to different extremes one at a time and analyzed.

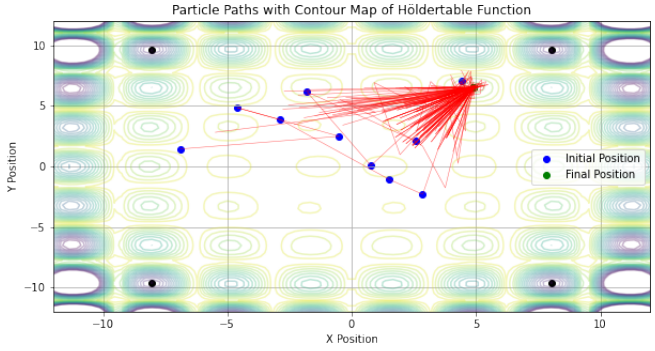
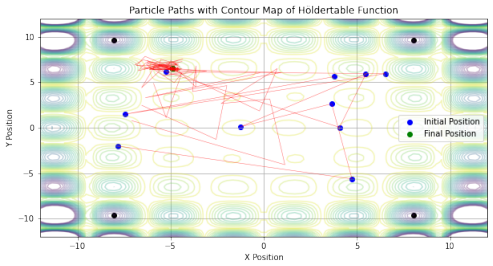
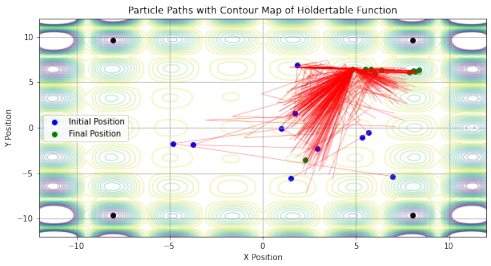
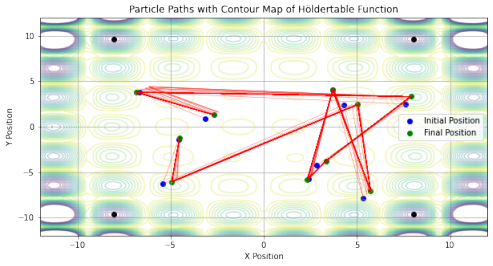
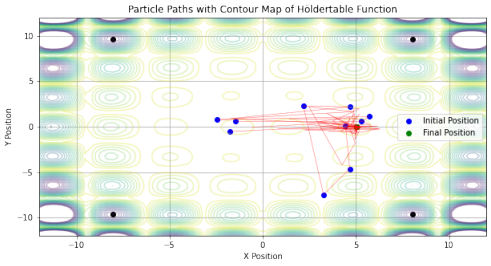
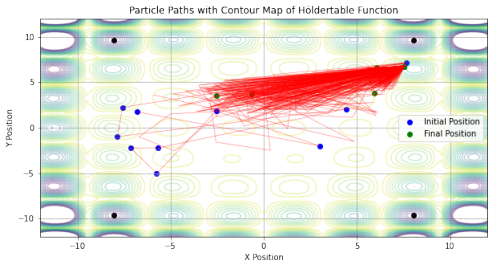
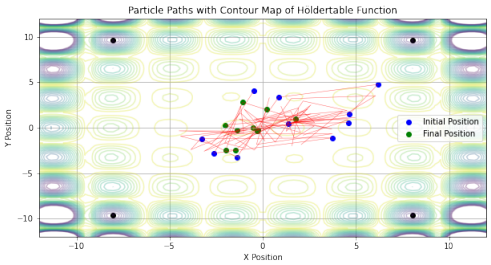
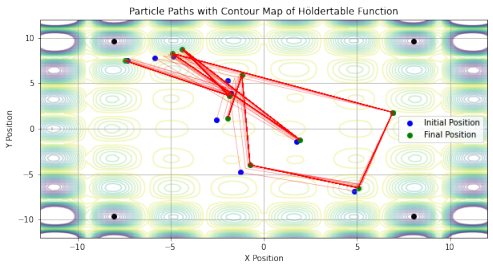
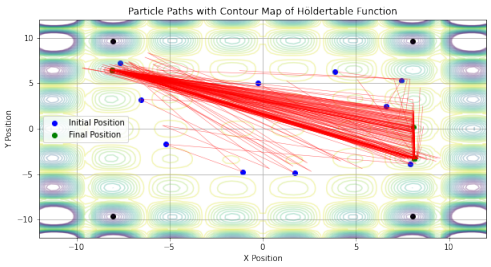
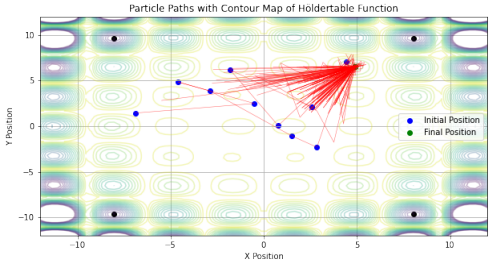
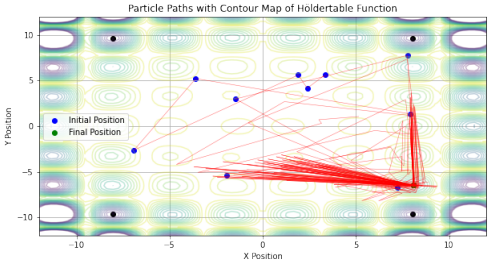


Figure 6: The resulting cost was cost = -4.712 with the position (4.90, 6.53). Note the optimal of -19.2085 at $(\pm 8.05502, \pm 9.66459)$

Table 2: Particle paths with various hyperparameter regimes

Parameter	**Value 1**	**Value 2**
c_1, cognitive	 $c_1 = 0$, best cost = -4.712 at (-4.90, 6.53)	 $c_1 = 1$, best cost = -9.37 at (8.04, 6.32)
c_2, social	 $c_2 = 0$, cost = -5.519 at (7.89, 3.33)	 $c_2 = 1$, cost = -1.733 at (5.02, -5.20)
w, inertial	 w = 0, cost = -7.93 at (7.55, 6.71)	 w = 1, cost = -1.703 at (-1.11, -0.073)
k, # of neighbors	 k = 1, cost = -6.356 at (-4.37, 8.79)	 k = 5, cost = -9.50 at (-8.10, 6.48)
p, Minkowski p-norm	 Equivalent to baseline: p=1, cost = -4.712 at (4.90, 6.53)	 p = 2, cost = -9.505 at (8.10, -6.48))

Note that $w = 1$ was used for the high extreme of w since a value of 2 led to particles flying off the bounds of the graph very often. This is a problem that the algorithm ran into quite often because of the small domain of the objective function. Even in the baseline parameter set, sometimes a particle would fly off the domain, resulting in a `ValueError`

In this bounded, multi-minima regime, one can draw more conclusions about the effect of hyperparameters

- c_1: Very similar to the Rosenbrock analysis. A low c_1 parameter means the particles follow the swarm tightly with very few red lines except at swarm optimal, and a high c_1 value shows lots of movement and longer red lines (in this case, it appears that the individual best locations ended up near the same location, hence the pull towards the same point)
- c_2: Very similar to the Rosenbrock analysis. A low c_2 parameter means the particles do not listen to swarm behavior and predictably follow their optimal best (hence the repeated, dark lines), while a high c_2 parameter increases the "pull" of the swarm toward the current swarm optimal
- w: Similar to Rosenbrock but with more problems. A low w tends to pull the swarm to the current optimum faster, while a high w causes the particles to fly off into space. It took over 20 tries with reduced iterations to make sure the particles stayed on the domain
- k: Similar to the Rosenbrock analysis. Again, there is a similar effect to the c_2 parameter, where having fewer neighbors creates dense, repeated paths for each particle, and having more neighbors increases social interactions toward the swarm optimal
- p: Not similar to the Rosenbrock analysis. Interestingly, there are more diagonal paths when $p = 1$ and more right-angle paths when $p = 2$. This is likely just due to noise, the randomness of the algorithm, and the visualization being a one-time experiment

Also note in 100 iterations, **NONE** of the hyperparameter regimes found the global minima. While some experiments found the closest local minima of -9.5 near $(\pm 8.10, \pm 6.48)$, the global minima of -19.209 were never reached. Also, note the high restart rate present due to the small domain. This means that Particle Swarm Optimization may require more computation (ex. more particles, more iterations) and may not be ideal for functions with small domains and many local minima without adapting the algorithm.

5 Application 3: Implementation from Scratch and Test with Eggholder Function

5.1 Implementation from Scratch

To fully understand how this algorithm works, the GlobalPSO version was implemented from scratch using Python and NumPy. The implementation was inspired by [Tam, 2021] and was modified for bounded-domain problems and visualization. By maintaining arrays for each particle's position, each particle's velocity, each particle's best position and cost, and the swarm's best position and cost, the equation from part 1 was able to be implemented. The implementation was relatively straightforward using standard NumPy and slicing operations.

The initialization of particle position was done with `np.random.uniform(min, max, (2, num_p))` and the initialization of particle velocities was done with `np.random.randn(2, num_p) * scalar`, where the scalar is dynamically set based on the domain of the objective function. As Tam (2021) explains, a normal distribution centered at 0 for particle velocity ensures a proper exploration space at the beginning of the algorithm [Tam, 2021].

5.2 Proof of Concept with Spherical Function

To make the algorithm is working as expected, it was tested on the spherical objective function in 2D space: $f(x, y) = x^2 + y^2$. With a testing domain of (-10, 10), the velocity scalar was set to 1 and the particles were initialized between $(\pm 10, \pm 10)$. Using the baseline parameters of $c_1 = c_2 = w = 0.5$, 100 iterations, and 10 particles, the minima are found.

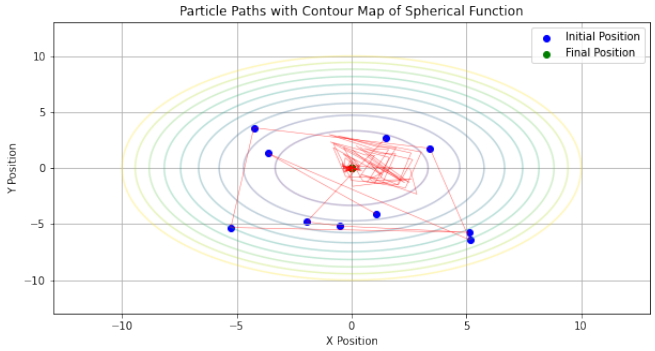


Figure 7: Particle Swarm Optimization on the spherical function reaches the optimal cost of 0 at (0,0)

5.3 Testing Performance with Eggholder Function

Now, this algorithm was applied to a much more complex function, the Eggholder function. This function has lots of sharp minima and the global minima of -959.6047 at (512, 404.2319) is located on the edge of the domain space of $[-512, 512]$ for x, y.

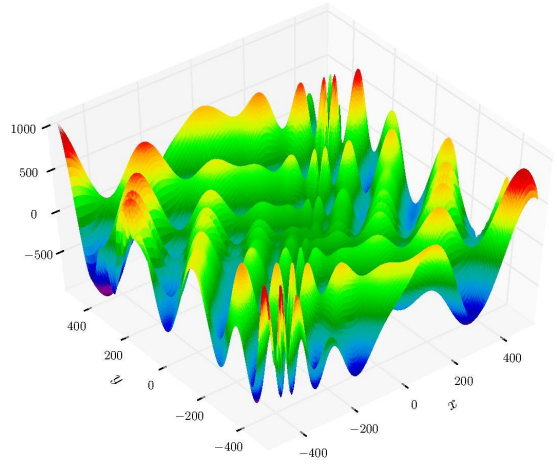
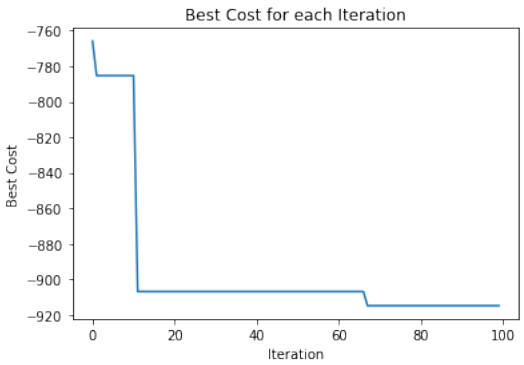
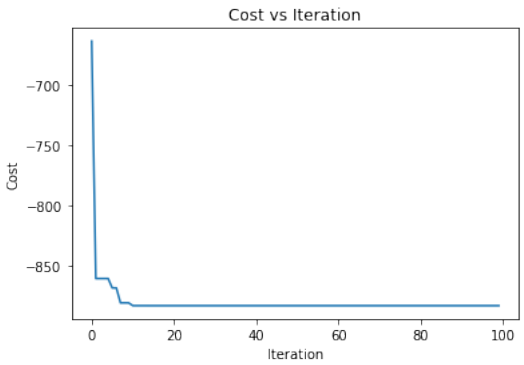


Figure 8: The Eggholder function, $f(x, y) = -(y+47)\cdot\sin\left(\sqrt{\left|\frac{x}{2}+(y+47)\right|}\right) - x\cdot\sin\left(\sqrt{|x-(y+47)|}\right)$
Figure from [Al-Roomi, 2015]

Given the complexity of this objective function, 50 particles (increased from the 10 used previously) were used with 100 iterations.

For the manual implementation, the positions are set to uniform random locations on the interval (-512, 512), and the velocity coefficient is set to 50 (since a domain of [-512, 512] is approximately 50 times the default of [-10, 10] used where the scalar is at its default of 1). This could also be tuned by the user to further refine the movement of the particle (akin to w). If a particle's calculated position is outside the domain space, the coordinate's value is clipped at 512 (or -512). For `PySwarms`, the locations were set on (-500, 500) to prevent too many `ValueError`s for out-of-bounds particles.

Table 3: Particle paths with various hyperparameter regimes

Metric	**By hand from Scratch**	**PySwarms**
Optimal Cost and Position (100 trials)	-959.641 at (-501.992, -413.739)	-894.579 at (-465.694 385.717)
Optimal Cost Progression		
Time taken (averaged over 100 trials)	4.974 milliseconds (Standard Deviation = 0.399 ms)	18.680 milliseconds (Standard Deviation = 1.925 ms)

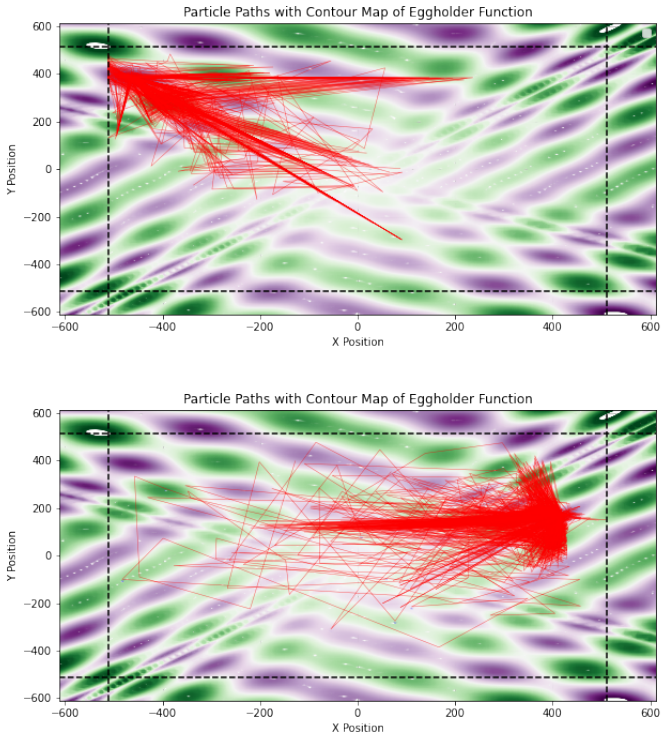


Figure 9: Comparing an example particle path of the by-hand manual implementation (top) with the PySwarms implementation (bottom)

Each of the metrics can be compared and contrasted

- Cost: The by-hand implementation finds a significantly better job at finding a lower-cost solution. This is likely due to the initial positions, which is a mixture of random chance, and the fact that the manual implementation allows for initialization between ± 500 and ± 512, which is closer to lower-cost minima.
- Cost function: Interestingly, the manual implementation still finds improvements after 50 iterations, showing a higher focus on exploration than optimization. The `PySwarms` implementation settles into its local minima by the 10th iteration and there is only marginal improvement
- Time: The manual implementation outperforms the `PySwarms` library by nearly 4x. This is likely due to the heavy overhead that the library implementation , which proves to be too bulky for an application like this but may be more effective in more complex problems (e.g., finding solutions to the Travelling Salesman problem, neural network hyperparameter space optimization).
- Visualization of particle paths. Again, the initial positions of the particles of the manual implementation can be set anywhere on the domain but are limited to be away from the edges on the `PySwarms` implementation. This likely creates a pull toward the lower-cost minima on the domain edge on the manual implementation but an emphasis on staying near the middle for the `PySwarms` implementation.

Overall, the implementation by hand does a better job of minimizing cost, exploring the space, and being time-efficient for this application. The manual implementation also allowed for custom modifications, as was shown with the position clipping.

5.4 Increasing Iterations and Particle Count to find Global Minima

Note that neither of the 100 attempts on the manual implementation and `PySwarms` implementation found the global minimum. To see if more computing power helps, the experiment was repeated using the manual implementation with 500 particles and 200 iterations. After 11 attempts (as the system is heavily dependent on initial locations), the global optimum was reached (total time, < 1 minute).

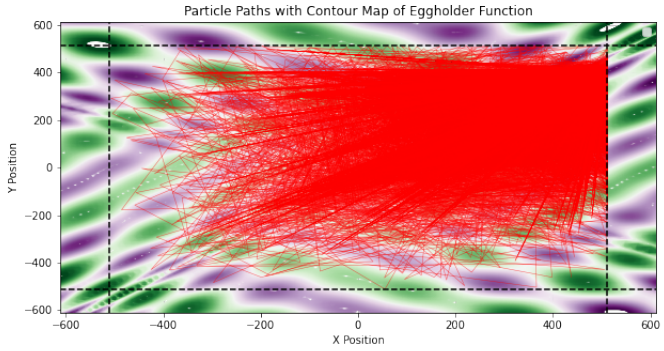


Figure 10: Visualization of 500 particles finding the optimal cost of -959.461 at (512, 404.232)

It is quite impressive that a simple swarm coded by hand in ≈ 20 lines can find the global minima in the Eggholder function, a complex function designed to trick optimization algorithms, in under a minute. Again, there is a strong dependence on initial conditions, and using a smarter initialization strategy (or perhaps integrating knowledge about the objective function), may cut down this time even further.

6 Conclusions and Future Work

Overall, Particle Swarm Optimization is an easy-to-use and incredibly powerful algorithm. Mathematically, it is compact and avoids complex gradient calculations completely.

From the analysis using `PySwarms` and the Rosenbrock function, the stochastic nature of this algorithm was seen first-hand in the hyperparameter RandomSearch. It looks like an inertial parameter $w \approx 0.5$ produces the best results, but c_1 and c_2 are more or less randomly distributed in the $[0, 1]$ space. However, there is a very clear effect in making the hyperparameters take on extreme values. Modulating c_2 and k is a knob on the impact of the swarm on each particle, c_1 is a knob on the independence of each particle to travel to its optimal, and w is tightly correlated with the amount of motion (high w = particles fly away).

These hyperparameter patterns were corroborated with the analysis using the Hölder-Table function, apart from the p parameter. Here, though, the algorithm struggles with multiple local minima and a tightly bounded domain. The latter proved to be especially difficult for ‘`PySwarms` to handle, inspiring a manual implementation.

This manual implementation was compared with `PySwarms` on the Eggholder function, a complex function with more local minima and the global minimum on the edge of the domain. Overall, the manual implementation had greater success at optimizing and with lower compute costs. While using only 10 particles and 100 iterations failed to find the global minimum after >100 tries, upgrading to 50 particles and 200 iterations did find the global optima quickly.

Future work focuses on applying PSO, both the library and manual algorithm, for more complex use cases, such as deep learning hyperparameter optimization and approximation of np-hard problems.

References

[Al-Roomi, 2015] Al-Roomi, A. R. (2015). Unconstrained Single-Objective Benchmark Functions Repository.

[Cleghorn and Engelbrecht, 2017] Cleghorn, C. W. and Engelbrecht, A. P. (2017). Particle swarm stability: A theoretical extension using the non-stagnate distribution assumption. *Swarm Intelligence*, 12(1):1–22.

[Engelbrecht, 2013] Engelbrecht, A. (2013). Particle swarm optimization: Global best or local best? In *2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence*, pages 124–135.

[Miranda, 2018] Miranda, L. J. (2018). Pyswarms: a research toolkit for particle swarm optimization in python. *Journal of Open Source Software*, 3(21):433.

[Saha, 2014] Saha, A. (2014). Artificial bee colony (abc) algorithm applied to slope- stability in searching the critical surface.

[Tam, 2021] Tam, A. (2021). A gentle introduction to particle swarm optimization. MachineLearningMastery.com.

[van den Bergh and Engelbrecht, 2010] van den Bergh, F. and Engelbrecht, A. P. (2010). A convergence proof for the particle swarm optimiser. *Fundamenta Informaticae*, 105(4):341–374.

In addition to the above references, the official Python documentation for `PySwarms`, `NumPy`, `Matplotlib`, and `Pandas` were used heavily throughout this paper. The source code can be found at this Github repository