Parallelizing Particle Swarm Optimization

An empirical analysis of using HIP and GPU Accelerators on Function Optimization

Chris Toukmaji

AM 148 - GPU Programming for Scientific Computations
June 6, 2022

1 Background Information

1.1 Overview

Particle Swarm Optimization (PSO) was introduced in 1995 by Dr. James Kennedy and Dr. Russell Eberhart [1]. PSO was inspired by the simulation of the social behavior of large groups of animals, such as flocks of birds or schools of fish, looking for food. In general, PSO consists of groups of different agents/particles (birds in our analogy) with different velocities trying to find the global minimum of a function (food in our analogy). Minimizing a function is especially important in areas such as deep learning. A significant portion of the training phase in a deep learning model consists of finding the minimum of a loss function. A loss function computes the difference between the model's predicted outcome and the true outcome. The output of the loss function is used to update the model's weights, or predictors.

The traditional approach of minimizing a loss function is gradient descent, where the weights are updated by taking the gradient of the loss function, and taking a step in the direction of the gradient. There are two main issues with traditional gradient descent:

1. The loss function must be differentiable in order to take the gradient.
2. Prone to getting stuck in local minima.

PSO does not use a gradient in its optimization, so it can be applied to non-differentiable functions, which solves the first issue of gradient descent. PSO is still vulnerable to getting stuck in a local mimina, but it lowers the chance of getting stuck in a local minima by introducing randomness in the initialization of particles. Each particle's initial position and velocity is randomized. Thus, PSO is often referred to as a stochastic algorithm.

Aside from PSO, other optimization techniques that improve upon the susceptibility of premature convergence in gradient descent are Stochastic Gradient Descent, Momentum, and Adaptive Learning Rates. In general, global convergence is not guaranteed in non-convex functions. It may occur by chance, but it is not guaranteed. Global convergence is guaranteed in convex functions, since typical optimization functions like gradient descent are guaranteed to converge to a local minima. In a convex function, there is only one minima. Thus, the minima of a convex function is its global minima.

In PSO, each particle's velocity is a summation of three vectors that gets updated at every time step. In ecological contexts, these vectors are referred to as 'Inertia', 'Cognitive/Personal Influence', and 'Social Influence'.

$$\text{New Velocity} = \text{Inertia} + \text{Cognitive Influence} + \text{Social Influence}$$

The 'Inertia' term contains an initially-randomized velocity vector. The 'Cognitive Influence' term computes the difference between the particle's current position and its all-time minimum position. The 'Social Influence' term computes the difference between the particle's current position and minimum position that the entire swarm found. Note: there must be at least two unique initial velocities in the set of all particles in order for PSO to converge. This was not explicitly stated in any of the literature I had read, but it is a necessary requirement for convergence (see Appendix 1).

In mathematical context, we represent the velocity update as such:

ω is the inertia constant
V_i^k is the velocity of the ith particle at time k
P_i^k is the best current position of the ith particle at time k
X_i^k is current position of the ith particle at time k
P_g^k is minimum found by all of the particles at time k

c_1, c_2 are acceleration constants - typically a hyperparameter that is tuned
r_1, r_2 are randomly-sampled constants chosen from a uniform distribution on $[0, 1]$ Then,

$$V_i^{k+1} = \omega V_i^k + c_1 r_1 (P_i^k - X_i^k) + c_2 r_2 (P_g^k - X_i^k) \tag{1}$$

We use the updated velocity vector to calculate the new position of the particle.

$$X_i^{k+1} = X_i^k + V_i^{k+1} \tag{2}$$

[2].
Typically, ω, c_1, c_2 are fine-tuned for each case. Previous research shows that

$$\omega > \frac{1}{2}(c_1 + c_2) - 1 \tag{3}$$

will guarantee convergent particle trajectories [4]. The optimal values are $\omega = 0.7298$, $c_1 = c_2 = 1.49618$ [5]. Tuning these hyperparameters is not the focus of this project, so we will be using these values in the velocity update.

1.2 Particle Swarm Optimization Algorithm

As outlined in the background, each particle makes use of information by other particles in the swarm, but the information that will be shared is not inherently sequential, allowing the optimization to be parallelized. More specifically, the information shared amongst the population is the 'Social Influence' term, or the lowest minimum point find amongst all of the particles. This term is updated if a particle has found a point that is smaller than the current minimum. Then, all the other particles used this updated minimum to better guide their future moves.

Algorithm 1 Global Best Particle Search Optimization Update [4] - Adapted As Needed

repeat
  for each particle $i = 1, ..., n_s$ **do**
    if $X_i^k < P_i^k$ **then**
      $P_i^k = X_i^k$
    end if
    if $P_i^k < P_g^k$ **then**
      $P_g^k = P_i^k$
    end if
  end for
  for each particle $i = 1, ..., n_s$ **do**
    update the velocity using equation (1)
    update the position using equation (2)
  end for

until stopping condition is true

The stopping condition is either time-steps (iterations) or if our convergence updates are below some threshold. The runtime complexity of Particle Search Optimization is $O(TPD)$ where
T = number of iterations
P = number of particles
D = number of dimensions.

2

2.1 Programming the Parallelization via GPU

In order to parallelize PSO, we must eliminate one of the contributing factors: either iterations, particles, or dimensions. Unfortunately, we cannot parallelize time (iterations), so we introduce parallelization in the number of particles or the number of dimensions. We can use a GPU to help by designating either a thread or block for each particle, depending on which factor we are optimizing for. We use shared memory to update the globally-sourced minimum since multiple threads will be reading from and writing to the globally-sourced minimum at all times.

I implemented a parallelized Particle Swarm Optimization in two different ways.

2.1.1 One Kernel Method

This method consisted of only using one kernel with *n_particles* many blocks and *n_dimensions* many threads. The block index is the particle position in the array of particles, and the thread index is the dimension within the particle. The velocity and position updates are independent so those can easily be parallelized. However, updating the value (i.e. taking each dimension and doing operations with them) require threads to be synchronized. Lastly, we update the global minimum if we have found a more optimal point, which can be parallelized. Each of these three steps is handled within the same kernel. This implementation is named as **HIP_Particle_One_Kernel.cpp** for reference, but it is not the HIP code that I timed and analyzed.

2.1.2 Three Kernel Method

This is the approach I took when for my timings and analyses. This approach essentially takes the steps from the One Kernel Method and divides them into three kernels: 'update', 'calculate', and 'compare'. 'Update' changes the velocity and position vectors for each particle and parallelizes the update of the dimensions as well. 'Calculate' uses the updated parameters from the 'Update' kernel to compute the function output and writes to memory. 'Compare' uses the output from the 'Update' kernel to update the local minimum found and writes to memory.
The main advantages to a method like this is the speedup in the 'update' kernel. While the 'calculate' and 'update' kernels have a (block,thread) size of (n_particles, n_dimensions), the 'calculate' kernel actually launches size (1, n_particles * n_dimensions). Then, the particle number is the thread index integer-divided by n_dimensions and the dimension number is the thread index modulo n_dimensions. This improves upon the One Kernel Method because each dimension of each particle can be updated within the same block.

2.2 Files

In my directory, you will find the following files:

HIP_Particle.cpp

This file contains the GPU implementation of Particle Swarm Optimization with 3 kernels.

HIP_Particle.h

Header file for HIP_Particle.cpp

Particle.cpp

CPU implementation of Particle Swarm Optimization.

Particle.h

Header file for Particle.cpp

Makefile

Compiles the source code and creates an executable.

ParticleSwarmOptimization.iypnb

iPython notebook for Proof-of-Concept model and for visualizations. Visualization code was taken from Dr. Jason Brownlee's PSO notes [7]. Selected visualizations are available in Appendix 2.

2.3 How to Run

Listing 1: HIP Compilation

```
module load hip
make
sbatch slurm
cat gpuTest.out
```

Listing 2: CPU Compilation

```
g++ Particle.cpp -o Particle
./Particle
```

Note: These files will output the timing and results for experiment (a), described below. All other experiments (except for (b)) require uncommenting the correct function in $f(x, y)$.

3 Evaluation

3.1 Function Choices

In order to analyze my PSO implementation, I assembled a set of the following evaluation functions, each chosen for a specific purpose.

(a) $f(x, y) = x^2 + y^2$

- (a) Baseline function to compare GPU/CPU. Known convex function with global minimum at f(0,0) = 0.

(b) $f(a, b, c, d, e, f) = a^2 + b^2 + c^2 + d^2 + e^2 + f^2$

- (a) Comparison of GPU/CPU implementations. Mainly exercises our PSO implementation on high-dimensional functions. The code submitted will run for 2 dimensions and labeled functions/lines can be uncommented to run this function.

(c) $f(x, y) = -cos(x) * cos(y) * exp(-((x - \pi)^2 + (y - \pi)^2))$,

- (a) Comparison of GPU/CPU implementations. a.k.a Eason's Function. Uniform at zero except for a global minimum at $f(\pi, \pi) = -1$. See Appendix 2 for a visualization. This ensures we can converge to points with non-zero minimums.

(d) $f(x, y) = \sqrt{x} + \sqrt{y}$

- (a) One of the advantages of PSO over traditional optimization algorithms is that it does not need a gradient. Thus, we can perform PSO on non-differentiable functions.

(e) $f(x, y) = \begin{cases} 0, & (x < 0) \wedge (y < 0) \\ 1, & else \end{cases}$

- (a) Custom-made 3D version of the Heavyside Step Function. Testing PSO on non-continuous functions.

(f) $f(x, y) = \begin{cases} 0, & (x < 0) \wedge (y < 0) \\ abs(x) + abs(y), & else \end{cases}$

- (a) Custom-made 3D version of the Rectified Linear Unit (ReLU) Function. Testing PSO on non-continuous functions.

3.2 Hyper-parameters

For reproducibility, I utilize the following hyper-parameters for all experiments unless otherwise noted.

- $srand(1)$
- $c1 = 1.49618$
- $c2 = 1.49618$
- $w = 0.7298$
- $r1 = rand()/RAND\_MAX$ // bounded by 1
- $r2 = rand()/RAND\_MAX$
- $n\_particles = 100$
- $X_i^0 = 10 * rand()/RAND\_MAX$ // bounded by 10
- $V_i^0 = 10 * rand()/RAND\_MAX$

3.3 Results

Each function tests a different purpose of Particle Swarm Optimization so there will be slight differences in their experimental setups. GPU timings were remotely calculated via the Lux Node with hipcc. CPU timings were calculated locally compiled via g++ on WSL2.

(a) $f(x, y) = x^2 + y^2$
For this experiment, I recorded the amount of time it took to converge to $f(x, y) = 0$, and the amount of iterations. This was done with a while loop (i.e. while $f(x, y) > 0$, update), so these results are when $f(x, y) = 0$. This is important to note because the values PSO finds for x, y are actually not 0, but very small floating point numbers whose sum of squares is too small to be represented in type double, so 0 is returned instead.

CPU Iterations	CPU Time	CPU Point Found	GPU Iterations	GPU Time	GPU Point Found
2683	0.038452 s	f(8.57933e-163, 7.2619e-166) = 0	2267	0.0935007 s	f(-8.04404e-163, 7.77293e-163) = 0

(b) $f(a, b, c, d, e, f) = a^2 + b^2 + c^2 + d^2 + e^2 + f^2$
Identical experimental procedure as (a).

CPU Iterations	CPU Time	CPU Point Found	GPU Iterations	GPU Time	GPU Point Found
3272	0.086311 s	f(1.43771e-162, 1.34087e-162, -1.22851e-162, -1.47835e-162, 3.9567e-163, -2.43993e-163) = 0	100000 (did not converge to 0)	4.58696 s	f(-1.06336e-05, -6.02452e-06, -5.81858e-06, 1.75226e-05, -1.48034e-05, -2.73672e-06) = 7.16896e-10

(c) $f(x, y) = -cos(x) * cos(y) * exp(-((x - \pi)^2 + (y - \pi)^2))$,
Identical experimental procedure as (a) except the function converges to -1.

CPU Iterations	CPU Time	CPU Point Found	GPU Iterations	GPU Time	GPU Point Found
119	0.002992 s	f(3.14159 , 3.14159) = -1	99	0.00413379 s	f(3.14159 , 3.14159) = -1

(d) $f(x, y) = \sqrt{x} + \sqrt{y}$

Identical experimental procedure as (a).

CPU Iterations	CPU Time	CPU Point Found	GPU Iterations	GPU Time	GPU Point Found
5281	0.060103 s	f(0 , 0) = 0	100000 (did not converge to 0)	4.32276 s	f(3.8461e-52, 6.25755e-20) = 2.50151e-10

(e) $f(x, y) = \begin{cases} 0, & (x < 0) \wedge (y < 0) \\ 1, & else \end{cases}$

Identical experimental procedure as (a).

CPU Iterations	CPU Time	CPU Point Found	GPU Iterations	GPU Time	GPU Point Found
3	1.9e-05 s	f(-0.446653, 3.89224) = 0	100000 (did not converge to 0)	3.40321 s	f(24.3915, 42.7236) = 1

Upon further analysis, the combination of PSO and the Heavyside function probably wasn't a good match. This combination is extremely sensitive to the initialization of points, and since our points are initialized as positive, they will remain positive unless the additive term carries a magnitude that is negative enough to bring the sum negative. Initially, I thought srand() would alleviate this issue, but since the CPU and GPU implementations were compiled on different computers, this could lead to different seed values, leading to different initialization and performance. A better choice for a non-continuous function would have a larger range of values for $f(x, y)$. See Experiment F.

(f) $f(x, y) = \begin{cases} 0, & (x < 0) \wedge (y < 0) \\ abs(x) + abs(y), & else \end{cases}$

Identical experimental procedure as (a).

CPU Iterations	CPU Time	CPU Point Found	GPU Iterations	GPU Time	GPU Point Found
2	1.7e-05 s	f(-1.9251, 4.47105) = 0	2	0.000118688 s	f(-2.19869, 2.67098) = 0

4 Conclusion

Whether it was the CPU or GPU implementation, PSO was able to converge to the convex, non-convex, non-differentiable, non-continuous, and high-dimensional functions described above. Many of these function classifications cannot be optimized with techniques such as gradient descent due to being non-differentiable. Unfortunately, PSO was still susceptible to getting stuck in local minimas and a slow convergence rate which I capped at 100,000 iterations.

4.1 Future Optimizations

Further iterations would include comparing the local minima found amongst various optimization algorithms (i.e. gradient descent), formalizing runtime and performance analysis of One/Three Kernel Methods, and refactoring the code such that increasing the number of dimensions is more accessible. As far as optimization, I would further investigate how to parallelize both the particles and the dimensions simultaneously.

4.2 Empirical Benefits of GPU

As illustrated in Experiments (a) and (c), I was able to use GPUs to decrease the number of iterations needed to reach the same threshold of convergence. However, there is a slight increase in time, so this is a tradeoff factor that must be evaluated depending on the function and use-case. I realize that the collection of functions I utilized may not have been complex enough for GPUs to take advantage of.

Appendix

1: Proof that Single-Repeated Initialization Position Vectors Converge to Non-Extrema Points

In real-world use cases, it is impossible for multiple objects to exist in the same position or physical space. However, in a theoretical application, this can be an initialization step that could easily be overlooked and cause issues. Below, I prove that we must have at least 2 particles with distinct position initialization in order to converge.

Claim: Particle Swarm Optimization will converge to a Random Non-Extrema Point for a set of particles without distinct position initialization.

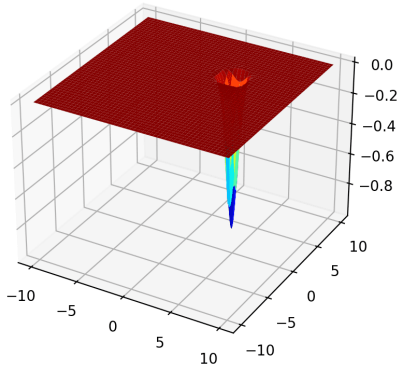
Proof:
Via Equation (1), we know that $V_i^{k+1} = \omega V_i^k + c_1 r_1 (P_i^k - X_i^k) + c_2 r_2 (P_g^k - X_i^k)$ where
$\{V_i^k, P_i^k, r_1, r_2\} \sim U(0,1)$,
$\omega = 0.7298$,
$c_1 = c_2 = 1.49618$.

If all particles have identical position initialization, then $|\{P^k\}| = 1$, so $P_i^k = P_g^k, \forall i \in n$. At initialization, we know that $P_i^k = X_i^k$. So,
$V_i^{k+1} = \omega V_i^k + c_1 r_1 (P_i^k - X_i^k) + c_2 r_2 (P_g^k - X_i^k)$
$V_i^{k+1} = \omega V_i^k + c_1 r_1 (P_i^k - X_i^k) + c_2 r_2 (P_i^k - X_i^k)$
$V_i^{k+1} = \omega V_i^k + c_1 r_1 (P_i^k - P_i^k) + c_2 r_2 (P_i^k - P_i^k)$
$V_i^{k+1} = \omega V_i^k$
$V_i^{k+1} = \omega^{k+1} V_i^0$
By Equation (2),
$X_i^{k+1} = X_i^k + \omega^{k+1} V_i^0$
$X_i^{k+1} = X_i^0 + \sum_{n=0}^{k+1} \omega^n V_i^0$
$X_i^{k+1} = X_i^0 + V_i^0 \sum_{n=1}^{k+1} \omega^n$
$X_i^{k+1} = X_i^0 + V_i^0 \sum_{n=1}^{k+1} \omega^n$
$X_i^{k+1} = X_i^0 + V_i^0 (\sum_{n=1}^{k+1} \omega^{n-1} + \omega^{k+1} - \omega^0)$
$\lim_{k\to\infty}(X_i^{k+1}) = X_i^0 + V_i^0 (\frac{1}{1-\omega} + \omega^{\infty+1} - \omega^0)$ because infinite geometric sum

$$\lim_{k\to\infty}(X_i^{k+1}) = \begin{cases} X_i^0 + V_i^0 (\frac{1}{1-\omega} - 1) & \text{if } |\omega| < 1 \\ \infty & \text{if } |\omega| \geq 1 \end{cases}$$

Thus, Particle Swarm Optimization will converge to infinity or a sum of the randomly initialized values depending on the value of ω.

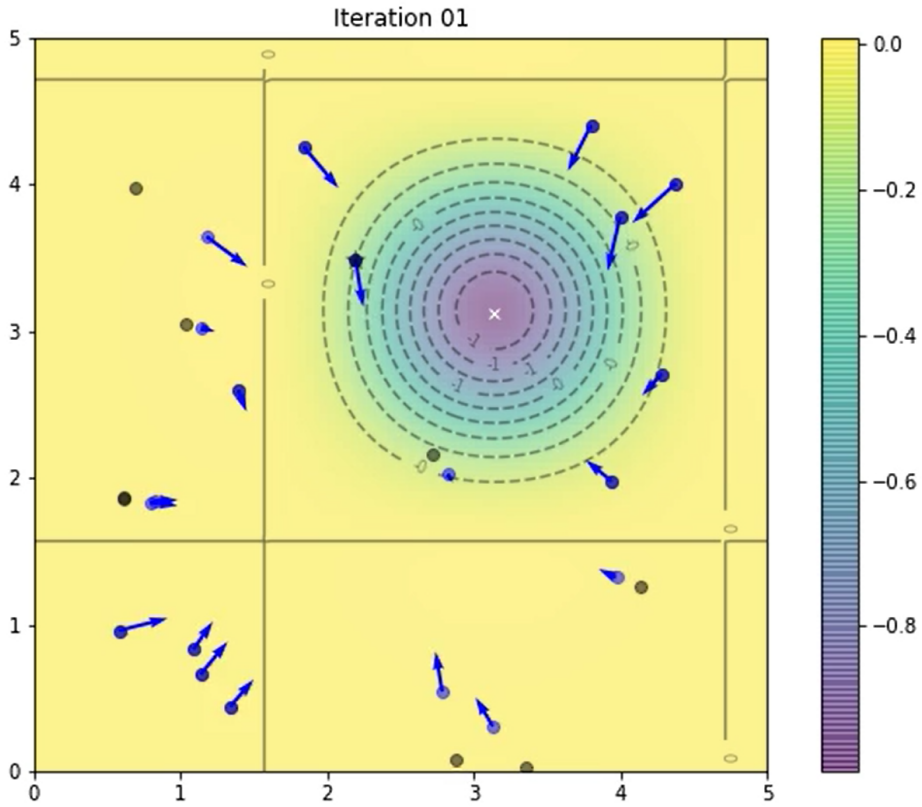
2: Easom's Function Plot

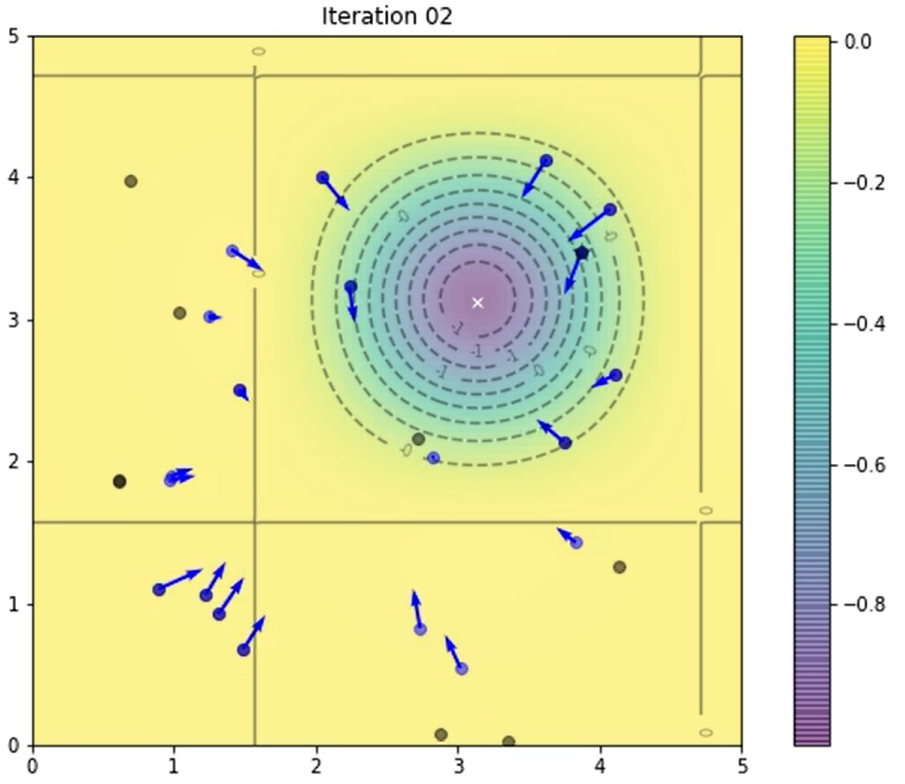


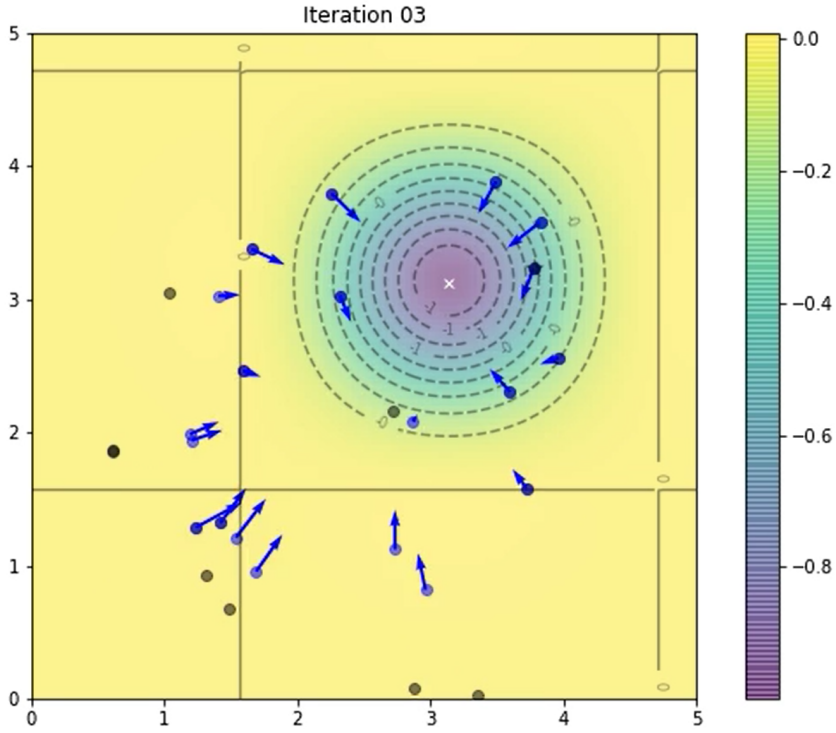
[8]

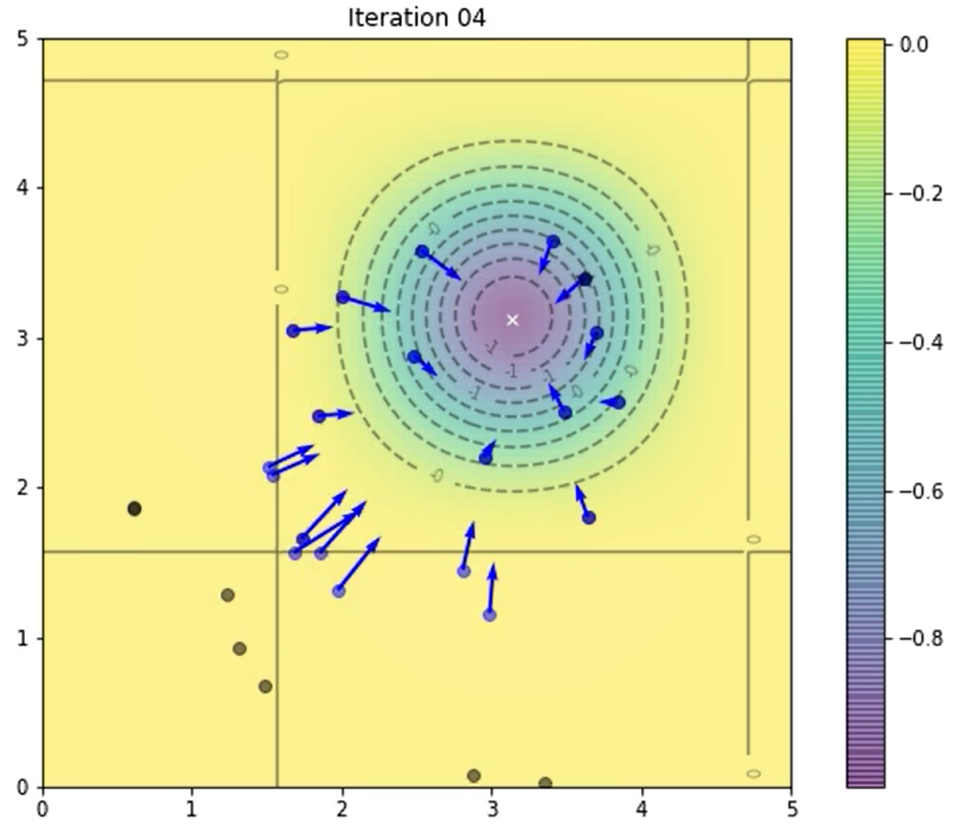
3: Particle Swarm Optimization Visualizations on Easom's Function

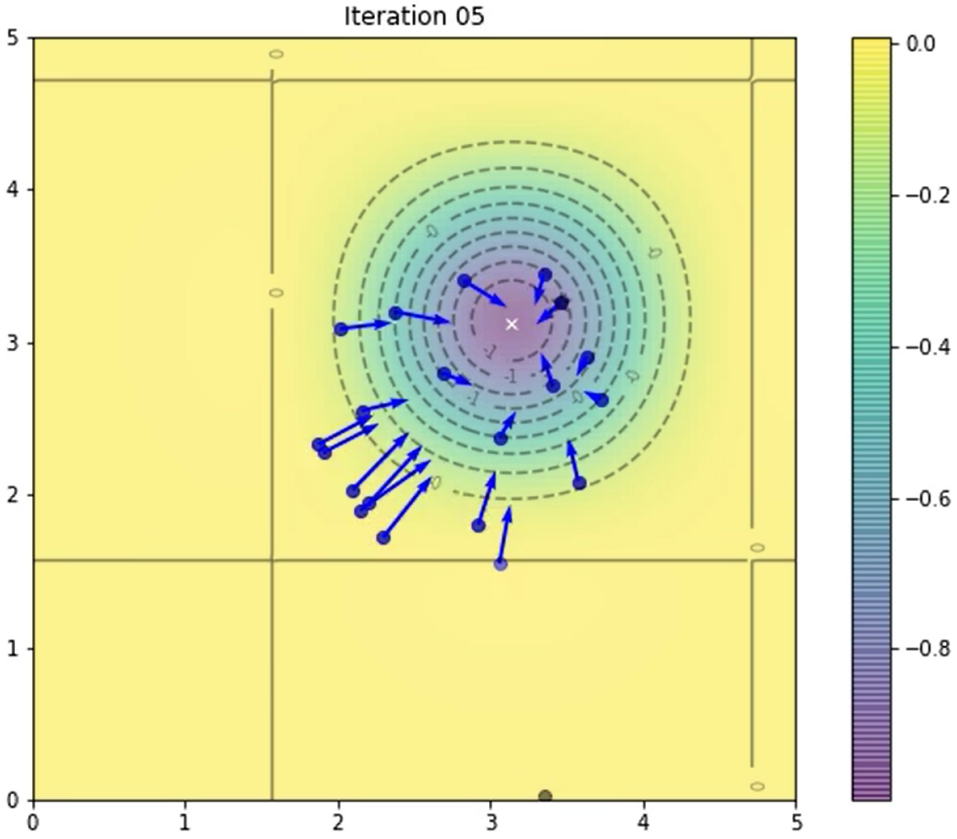
Visualizations generated using **ParticleSwarmOptimization.ipynb** [7].

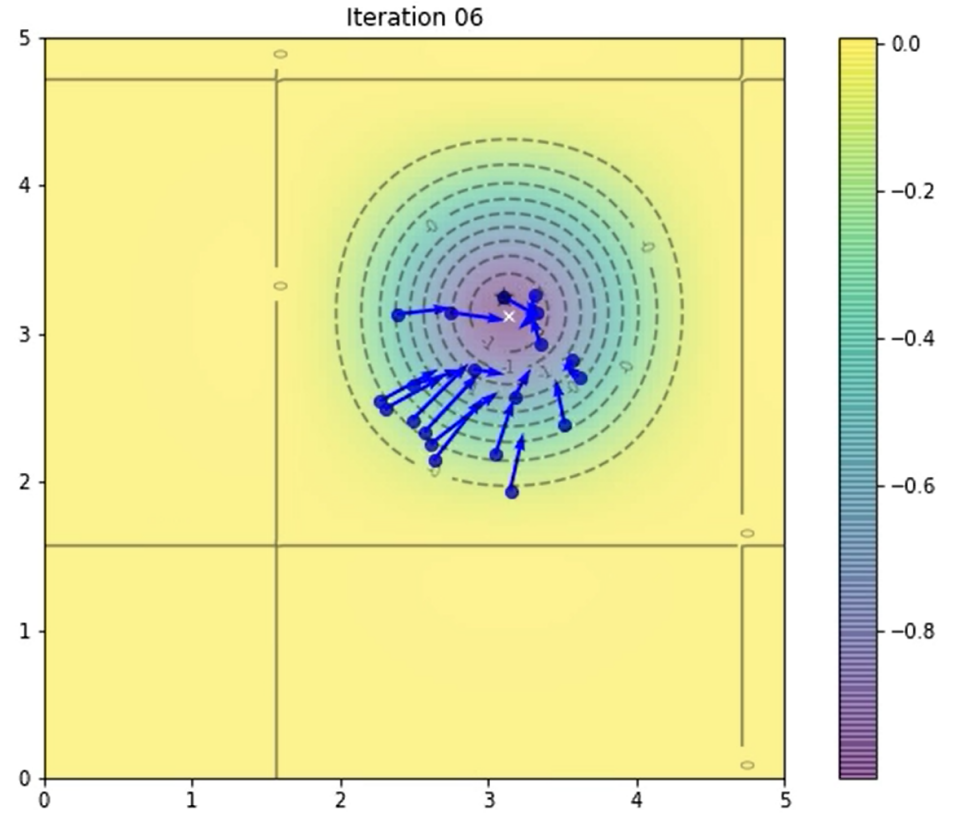


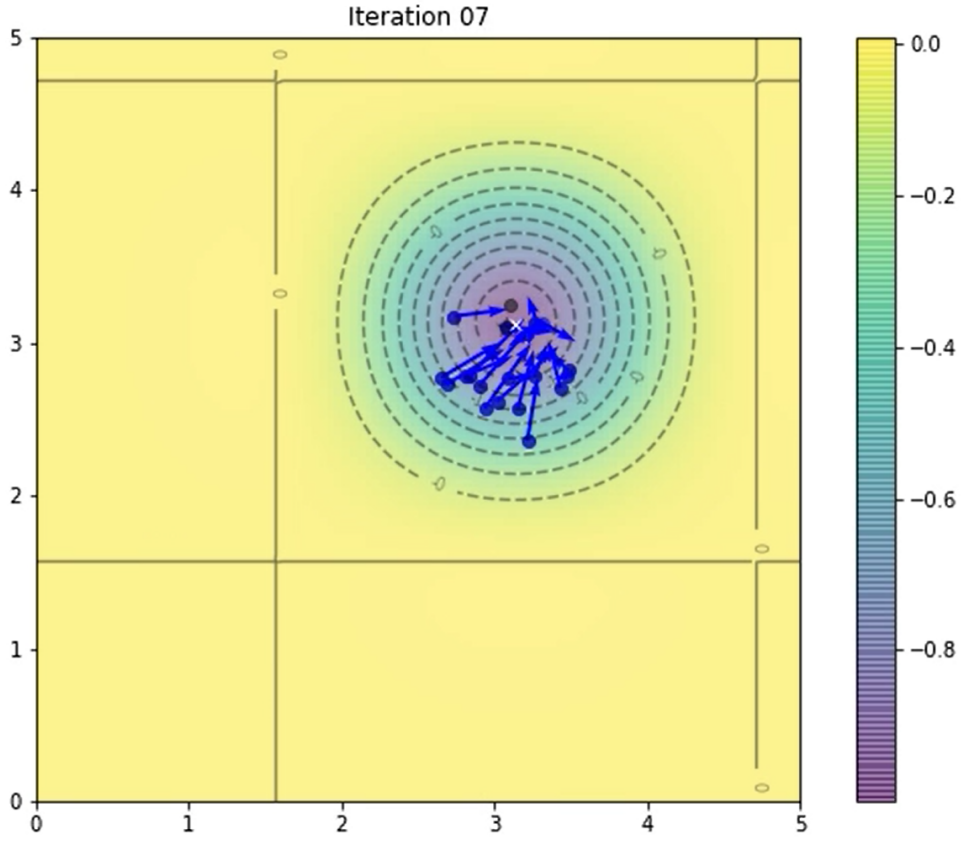

Iteration 02
0
1
2
3
4
5
0.0
−0.2
−0.4
−0.6
−0.8

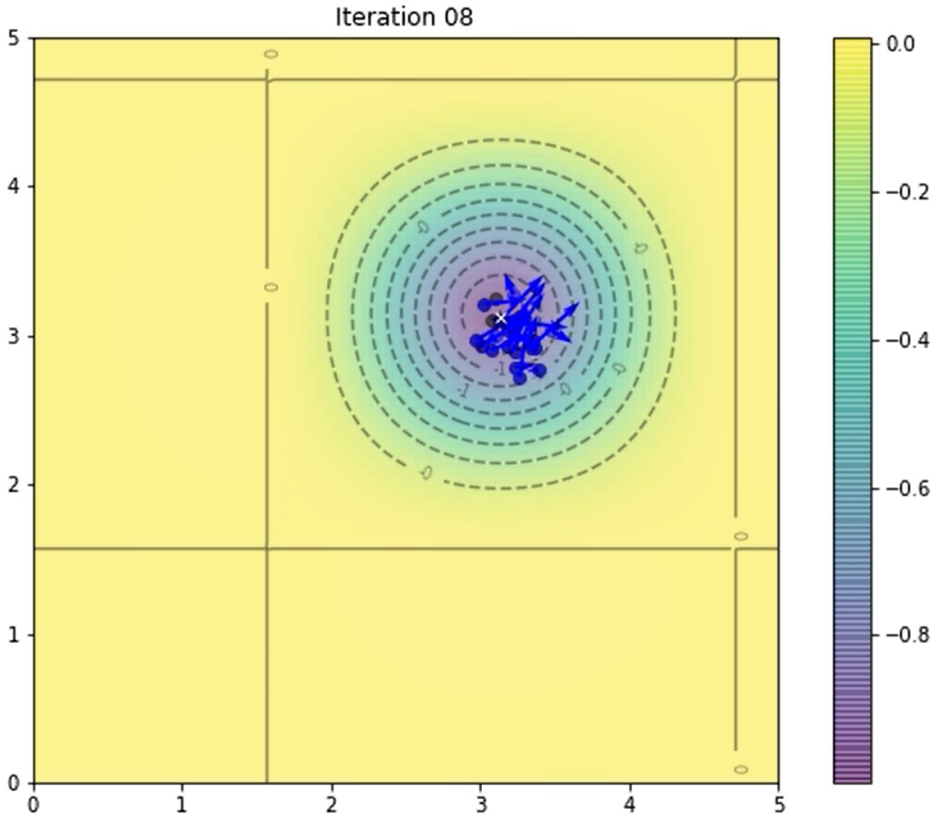

Iteration 03
0
1
2
3
4
5
0.0
−0.2
−0.4
−0.6
−0.8

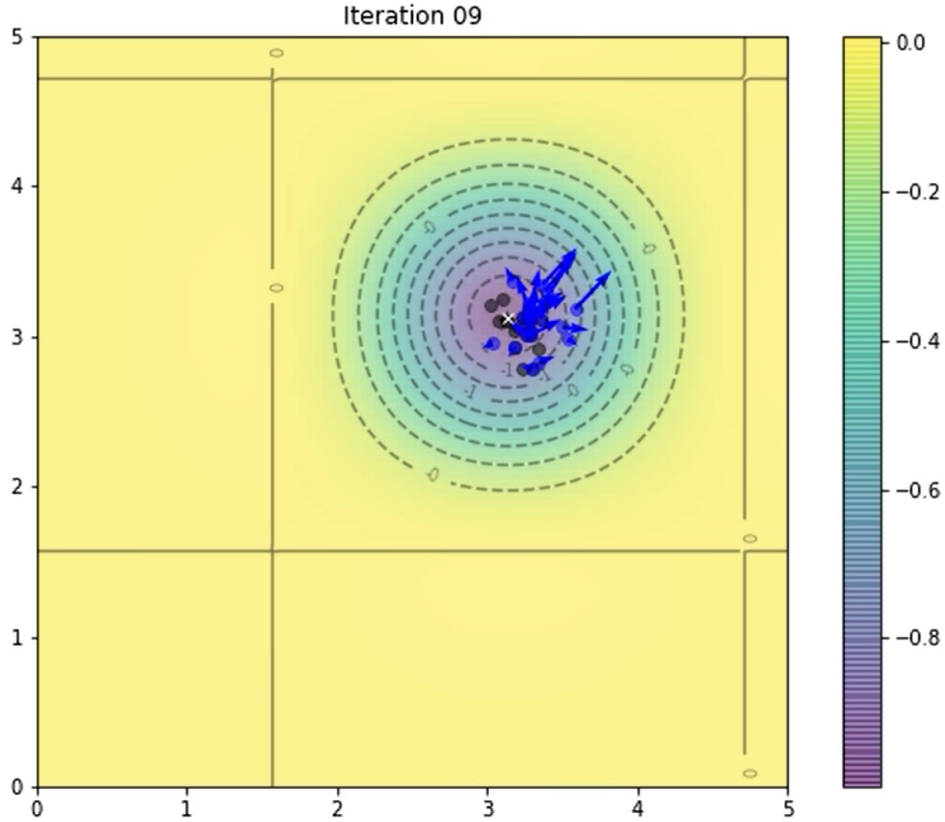

Iteration 04
0.0
−0.2
−0.4
−0.6
−0.8

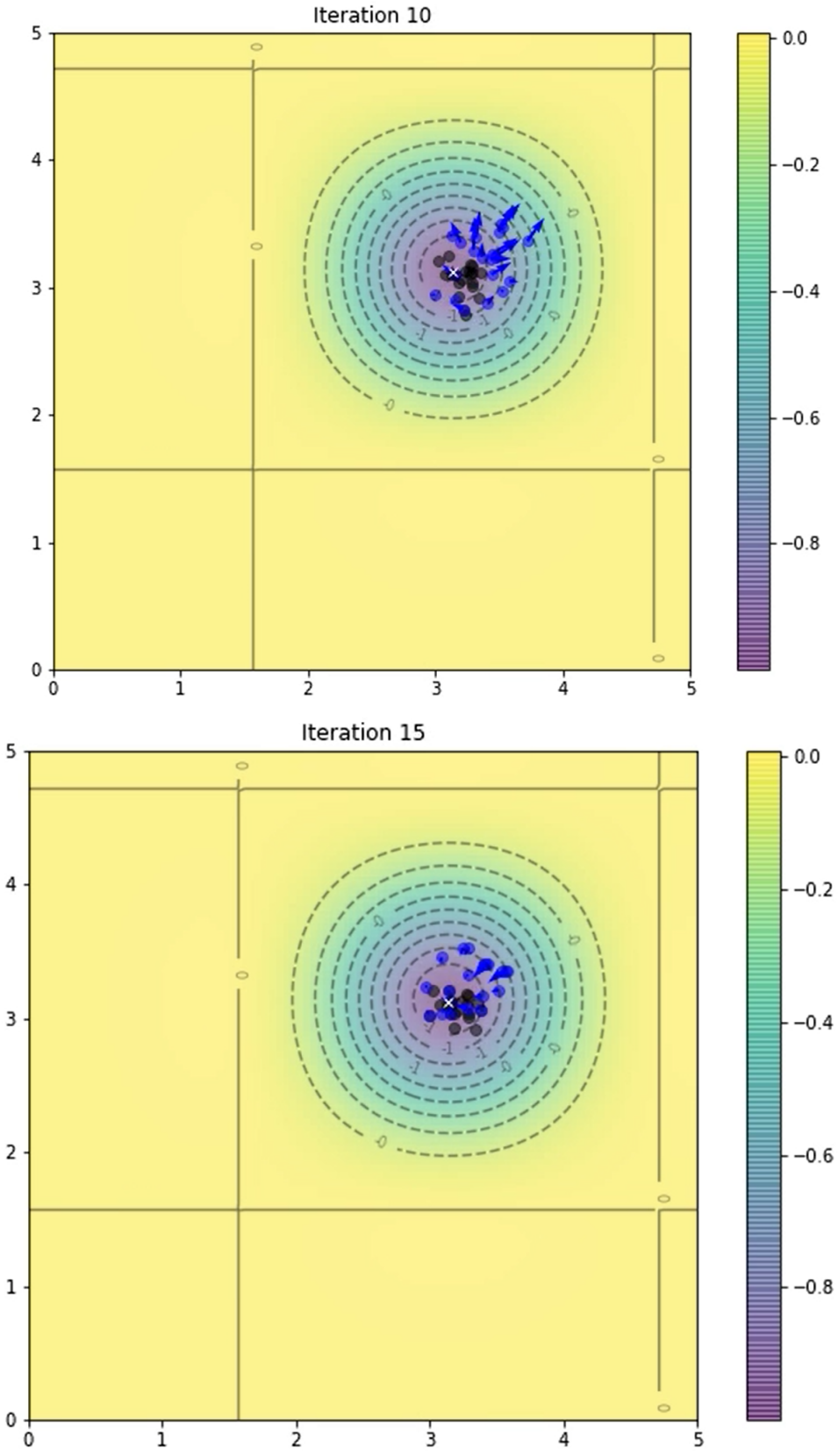

Iteration 05
0.0
−0.2
−0.4
−0.6
−0.8

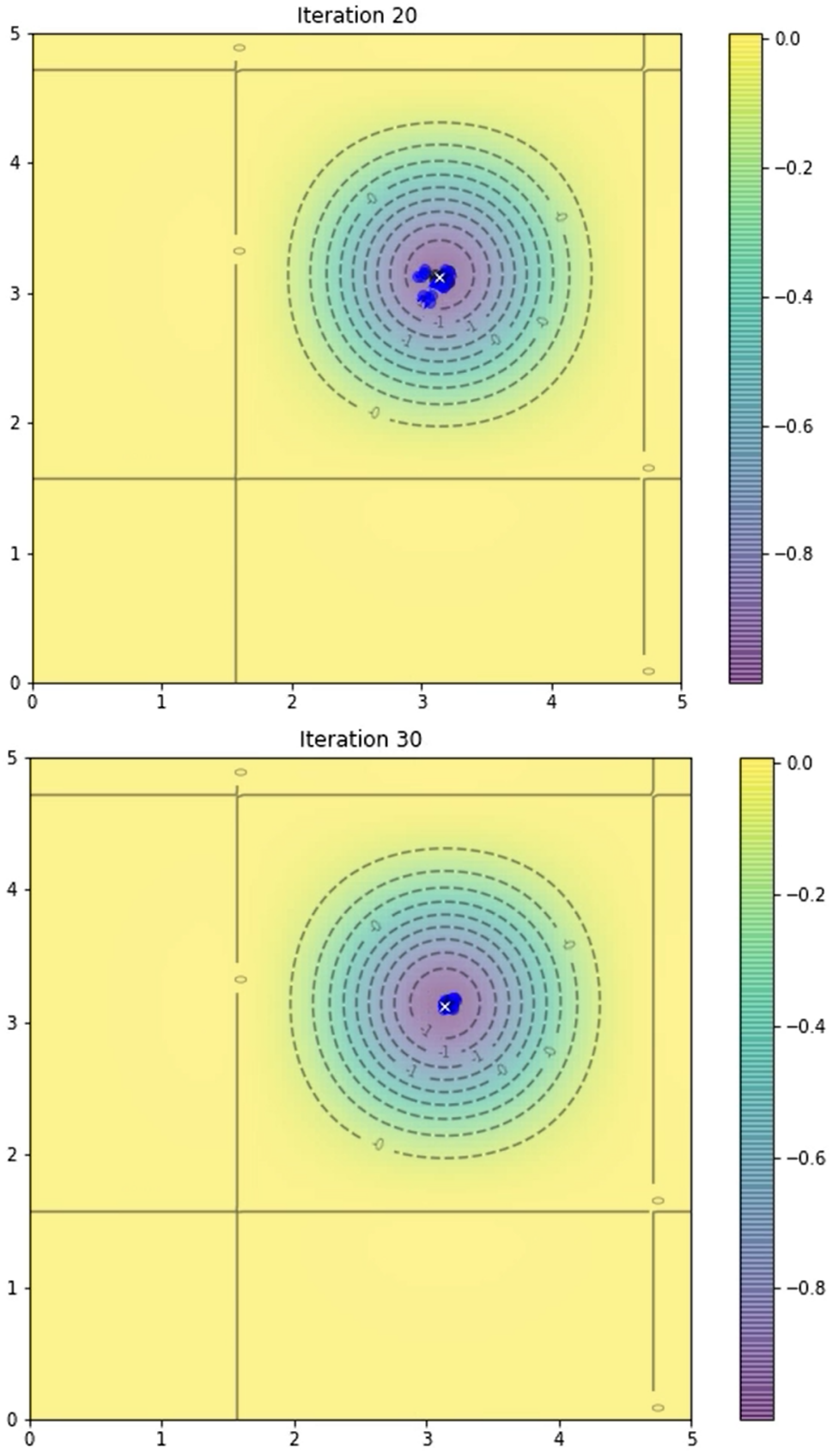

Iteration 06
0.0
−0.2
−0.4
−0.6
−0.8
0
1
2
3
4
5

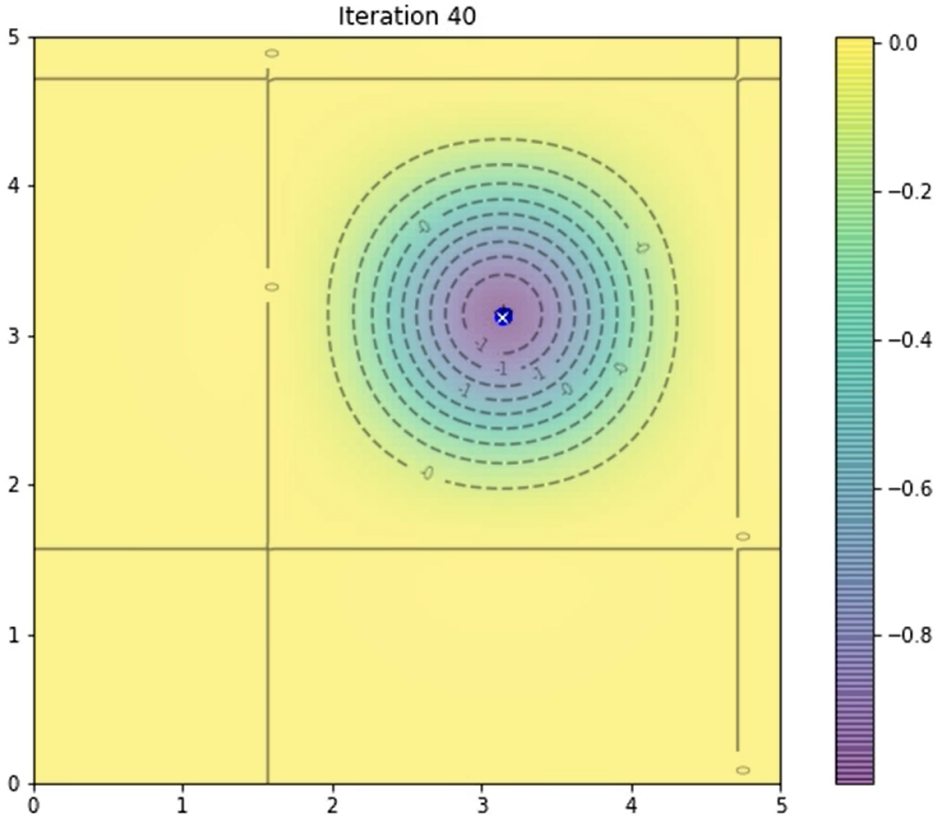

Iteration 07
0.0
−0.2
−0.4
−0.6
−0.8
0
1
2
3
4
5


Iteration 08
0.0
−0.2
−0.4
−0.6
−0.8


Iteration 09
0.0
−0.2
−0.4
−0.6
−0.8


Iteration 10
Iteration 15
0.0
−0.2
−0.4
−0.6
−0.8


Iteration 20
Iteration 30
0.0
−0.2
−0.4
−0.6
−0.8
0
1
2
3
4
5



Sources

[1]: https://ieeexplore.ieee.org/document/488968

[2]: https://onlinelibrary.wiley.com/doi/10.1002/eng2.12048

[3]: https://www.cil.pku.edu.cn/docs/20190117151926032330.pdf

[4]: https://web2.qatar.cmu.edu/ gdicaro/15382/additional/CompIntelligence-Engelbrecht-ch16.pdf

[5]: https://www.sciencedirect.com/science/article/abs/pii/S0020025505000630

[6]: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.833.8220rep=rep1type=pdf

[7]: https://machinelearningmastery.com/a-gentle-introduction-to-particle-swarm-optimization/

[8]: https://machinelearningmastery.com/2d-test-functions-for-function-optimization/