Particle Swarm Optimization in Matlab

Computational Intelligence - University of Nebraska

Zachary Swanson
November 21, 2024

Contents

Abstract

This report explores the implementation and analysis of a Particle Swarm Optimization (PSO) algorithm to minimize the six-hump camelback function, a non-linear and multi-variate optimization problem. The PSO algorithm was programmed in MATLAB using a modular design, with individual particles represented as objects that track their positions, velocities, and personal bests, while a swarm class managed the collective behavior and global best tracking. The algorithm's performance was evaluated by varying key parameters, including the number of particles, velocity limits, inertia weight, and acceleration constants, to assess their influence on convergence speed, solution quality, and swarm dynamics.

Results from the baseline configuration demonstrated the algorithm's ability to find the global minimum, with the first particle converging after 31 iterations and all particles achieving the global best after 166 iterations. Increasing the number of particles improved the convergence speed of the first particle but had minimal effect on the time required for all particles to converge. Velocity limits affected the swarm's ability to explore and exploit the search space effectively, with excessively restrictive limits hindering convergence. Moderate inertia weights led to the fastest convergence, while higher values caused overshooting and instability. Finally, the balance between cognitive and social acceleration constants was critical, with social influence essential for global convergence and excessive self-reliance delaying swarm-wide stabilization.

The study highlights the importance of parameter tuning in PSO to achieve robust performance and provides insights into how these parameters influence swarm behavior. The findings offer practical guidance for optimizing PSO configurations for complex optimization tasks.

1 Introduction

Particle Swarm Optimization (PSO) is a population-based stochastic optimization algorithm inspired by the collective social behaviors observed in natural phenomena, such as bird flocking and fish schooling. Proposed by Eberhart and Kennedy in 1995, PSO optimizes problems by iteratively improving candidate solutions based on their fitness within a multidimensional search space. A particle in the swarm represents a potential solution, characterized by its position and velocity, which are updated at each iteration based on its personal best position (P_{best}) and the global best position (G_{best}) of the swarm [1].

PSO is an iterative algorithm, where the position of every particle is updated with each iteration. The position update is governed by the equation

$$x_i(k+1) = x_i(k) + v_i(k+1), \tag{1}$$

where x_i is the position of particle i, v_i is the velocity of particle i, and k is the iteration number. The velocity is also updated at each iteration and the update is given by

$$v_i(k+1) = w \cdot v_i(k) + c_1\phi_1 \cdot (P_{best} - x_i(k)) + c_2\phi_2 \cdot (G_{best} - x_i(k)), \tag{2}$$

where w is the inertia constant, c_1 is the cognitive acceleration constant, c_2 is the social acceleration constant, and $\phi_1, \phi_2\ U(0,1)$ are uniformly distributed random variables that introduce stochasticity. Each component of Equation 2 has a unique effect on the particle's journey through the search space [1].

The first term, $w \cdot v_i(k)$, governs the persistence of a particle's velocity, enabling it to maintain its momentum across iterations. A higher inertia constant encourages exploration by allowing particles to traverse broader areas of the search space, while a lower inertia constant promotes exploitation by focusing on refining solutions near promising regions.

The second term, $c_1\phi_1 \cdot (P_{best} - x_i(k))$, reflects a particle's self-awareness and its tendency to return to its own best-known position (P_{best}). It ensures that a particle prioritizes its individual success in the optimization process. The cognitive acceleration coefficient c_1 controls the strength of this pull.

The final term, $c_2\phi_2 \cdot (G_{best} - x_i(k))$, embodies the particle's inclination to learn from the swarm by moving toward the global best position (G_{best}). The social acceleration coefficient c_1 determines the intensity of this social influence.

A fitness function is necessary to evaluate the quality of each particle's position in the search space, guiding the optimization process. The personal best position (P_{best}) and the global best position (G_{best}) of the swarm are determined based on the best values of the fitness function achieved at given point. For minimization problems, the fitness value reflects how close the particle is to the optimal solution, with lower values indicating better solutions. In this project, the six-hump camelback function serves as the fitness function

$$z = \left(4 - 2.1x^2 + \frac{x^4}{2}\right) \cdot x^2 + xy + \left(-4 + 4y^2\right) \cdot y^2, \tag{3}$$

where $x, y \in [-5, 5]$.

Velocity limits and swarm topologies are other important topics related to PSO. Velocity limits control the maximum allowable speed at which particles can traverse the search space. Capping the velocity helps prevent particles from overshooting optimal regions and ensures that they explore the solution space in a controlled manner. The communication structure, or topology, of the swarm dictates how particles interact with one another. Two common topologies are global best and local best. For global best, every particle in the swarm considers the global best position (G_{best}) as a reference point. This promotes rapid convergence but increases the risk of getting trapped in local minima due to a lack of diversity. For local best, each particle interacts with a subset of the swarm, typically its neighbors in a ring or wheel topology. This fosters diversity and helps prevent premature convergence, albeit at the cost of slower convergence rates [1].

The goals of this experiment include identifying the global minimum of the six-hump camelback function in Equation 3 and analyzing the effects of varying PSO parameters including the number of particles, inertia constant, velocity limits, and acceleration constants, on the convergence behavior.

2 Methodology

The PSO algorithm was programmed using two classes in Matlab: *particle* and *swarm*. The particle class represented a single particle in the swarm and was responsible for keeping track of and updating its current and personal best positions and its velocity. Equations 1 and 2 were implemented in the particle class' *update(·)* method. The class also stored a fitness function callback that evaluated the value of the and updated the P_{best} value as necessary with each call of the *update(·)* method. The class also had *threshold* and *no_progress_count* parameters that allowed the class to provide a metric of whether the particle had settled somewhere and how long it had been there.

The swarm class managed a swarm of N particle objects and randomly initialized the position of each particle of the problem space, i.e. $x, y \in [-5, 5]$. The swarm class also contained the w, c_1, and c_2 constants and the velocity limit which were the same for each particle. The swarm class was also responsible for monitoring and updating the G_{best} position. The class feature a *run(·)* method that would run up to a specified number of iterations, e.g. 1000. However, the swarm class also had a *patience* parameter that specified how many iterations the particles should be "stationary", i.e. moving under the threshold, before declaring no progress and terminating the optimization. This related back to the *threshold* and *no_progress_count* parameters of the particle class and allowed tracking when the swarm reached a stable state. The swarm class also tracked the history of the positions and fitness function values of each particle at each iteration. This allowed analysis of when the swarm first reached the minima and if/when the remaining particles reached the minima also.

The implemented PSO algorithm was used to optimize the six-hump camelback function in Equation 3, and a series of experiments were conducted to analyze the effects of various algorithmic parameters on performance. The primary goal was to determine how changes in particle count, velocity limits, inertia weights, and acceleration constants influenced convergence speed, solution quality, and swarm stability. Each experiment is described below. For all experiments, only the G_{best} topology was used as it performed sufficiently well for the purposes of this experiment.

2.1 Baseline Experiment

A baseline configuration was established with 30 particles, an inertia weight $w = 0.5$, acceleration constants $c_1 = c_2 = 1.5$, a velocity limit $v_{\max} = 2.0 = 0.2(x_{\max} - x_{\min})$, and a maximum iteration count of 1000. The patience parameter was set to 20 iterations, meaning the algorithm would terminate early if all particles were stationary for 20 consecutive iterations. This configuration served as a reference for comparison across experiments.

2.2 Effect of Particle Count

The number of particles in the swarm was varied to observe its impact on convergence and solution quality. Swarm sizes of $N = 1 to 50$ were tested, while all other parameters were kept consistent with the baseline configuration. Metrics such as the number of iterations required to converge and the consistency of the global best solution were recorded.

2.3 Effect of Velocity Limits

The velocity limit, $v_{\max}$, was altered to evaluate its role in controlling particle movement and avoiding divergence. Experiments were conducted with no velocity limit and velocity limits between 1% and 50% of the problem space, i.e. $[-5, 5]$. Hence, the velocity limits ranged from 0.1 to 5. The results were compared to the baseline to assess how different limits influenced exploration and exploitation.

2.4 Effect of Inertia Weight

The inertia weight w was adjusted to investigate its effect on swarm dynamics. A range of values were tested from 0 to 1.0 in steps of 0.1. The results were analyzed to determine how w impacted the swarm's ability to explore the search space and converge on the global minimum.

2.5 Effect of Acceleration Constants

The cognitive and social acceleration constants, c_1 and c_2, were varied to examine their influence on particle behavior. Each constant was varied in steps of 0.5 over the range 0 to 2.0 such that there were scenarios with varying degrees of reduction on self-exploration and enhancement of social influence and vice-versa. This helped analyze the trade-offs between local refinement and global convergence.

2.6 Performance Metrics and Analysis

For each configuration, the following metrics were recorded:

- **Convergence Speed**: Number of iterations required for the swarm to stabilize and identify the global minimum.
- **Solution Quality**: Fitness value at the global best position.
- **Swarm Dynamics**: The distribution of particles around the global minimum over time, tracked using the positional history of particles.

The experimental results were visualized using fitness plots and convergence curves. Comparisons between configurations were made to assess the sensitivity of the PSO algorithm to parameter variations and provide recommendations for selecting optimal parameter settings.

3 Results

The baseline model had a particle achieve the global best position of (-0.089842, 0.712656) after 31 iterations which produced the global minimum value of -1.03163. All particles achieved the global minimum taking 166 iterations for the final particle to settle into the global minimum.

3.1 Effect of Particle Count

For 1-3 particles the global minimum was not discovered. This suggests that there was not enough social information to sufficiently explore the search space leading to suboptimal performance. For 4 to 50 particles, the swarm was able to find the global minimum and have all particles reach the minimum. Figure 1 shows the number of iterations required for the first particle and for all particles to converge to the global minimum as a function of the number of particles. There are some outlier results where the convergence takes longer than the trend suggests like with 16 particles. These outliers are likely a result of where the particles were initialized. Regardless, the results show an inverse relationship between the number of particles and the number of iterations for the first particle to converge to the global minimum. This is a reasonable result because more particles (a) increases the probability of at least one particle being initialized near the best position and (b) increases the coverage of the search space. However, the number of particles did not seem to produce a significant change in the number of iterations for all particles to converge.

3.2 Effect of Velocity Limits

The swarm had at least one particle achieve the global minimum for all velocity limits evaluated. This suggests that the other settings of the baseline model produced a fairly stable swarm.

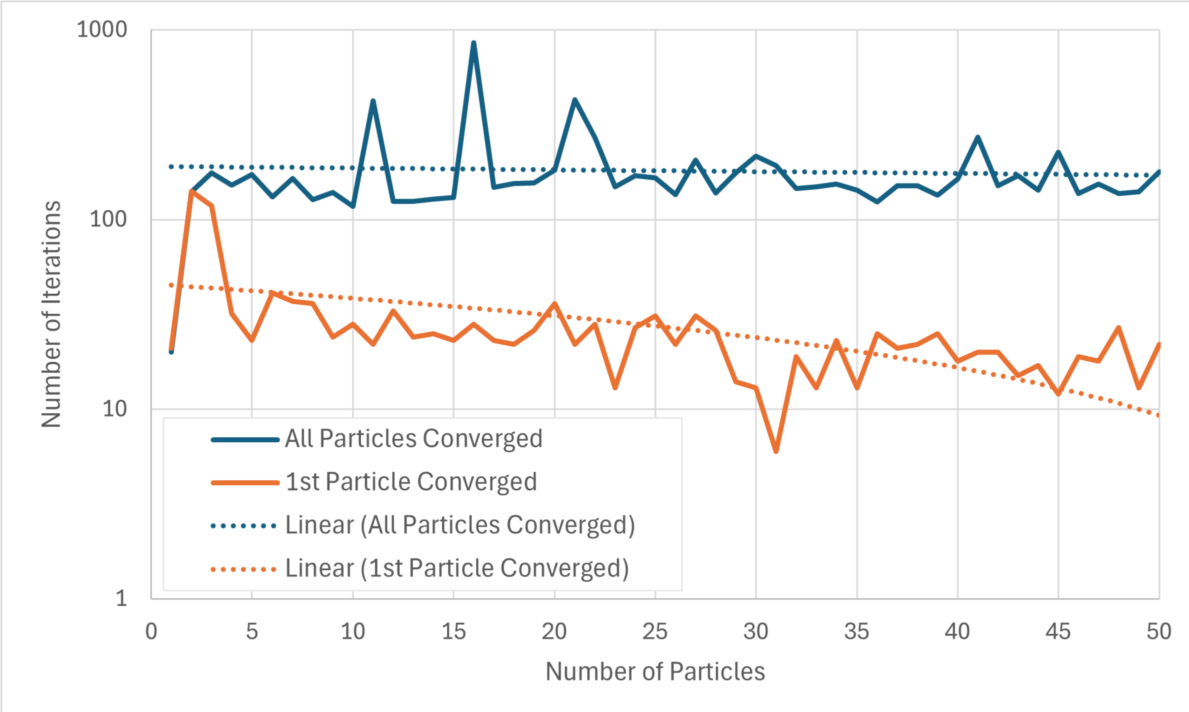

Figure 1: Plot demonstrating the effect of the number of particles on the number of iterations for the first particle to converge and on the number of iterations for all particles to converge.

Otherwise, the effects of the velocity limit may have been more pronounced and unbounded jumps across the search space may have been observed. In addition to the convergence speed of the first and all particles, Figure 2 also shows the average fitness value of all particles at the end of the simulation. This was plotted because at least one particle achieved the global minimum for all tests. For velocity limits less then 0.3 and the outlier at 0.7, not all particles converged. This may be attributed to the limited size of step the particle could take each iteration. As a result, initially distant particles would not have the velocity to reach the G_best position in the allotted 1000 iterations.

3.3 Effect of Inertia Weight

Again, the swarm had at least one particle achieve the global minimum for all inertia weights evaluated. However, not all particles were able to converge to the global minimum for $w \geq 0.8$. At that level of inertia, particles that are being pulled in from larger distances are building more momentum as they come. As a result, these particles are more likely to continually overshoot the global minimum. Also, swarms with $w \in [0.1, 0.3]$ achieved the fastest convergence of all the experiments conducted across all the parameters. At these levels of inertia with other baseline parameters, the first particle reached the global minimum in less than ten iterations and all particles converged in less than 50 iterations. Figure 3 shows these converge speeds. The figure demonstrates a clear relationship between the inertia weight and the speed of convergence.

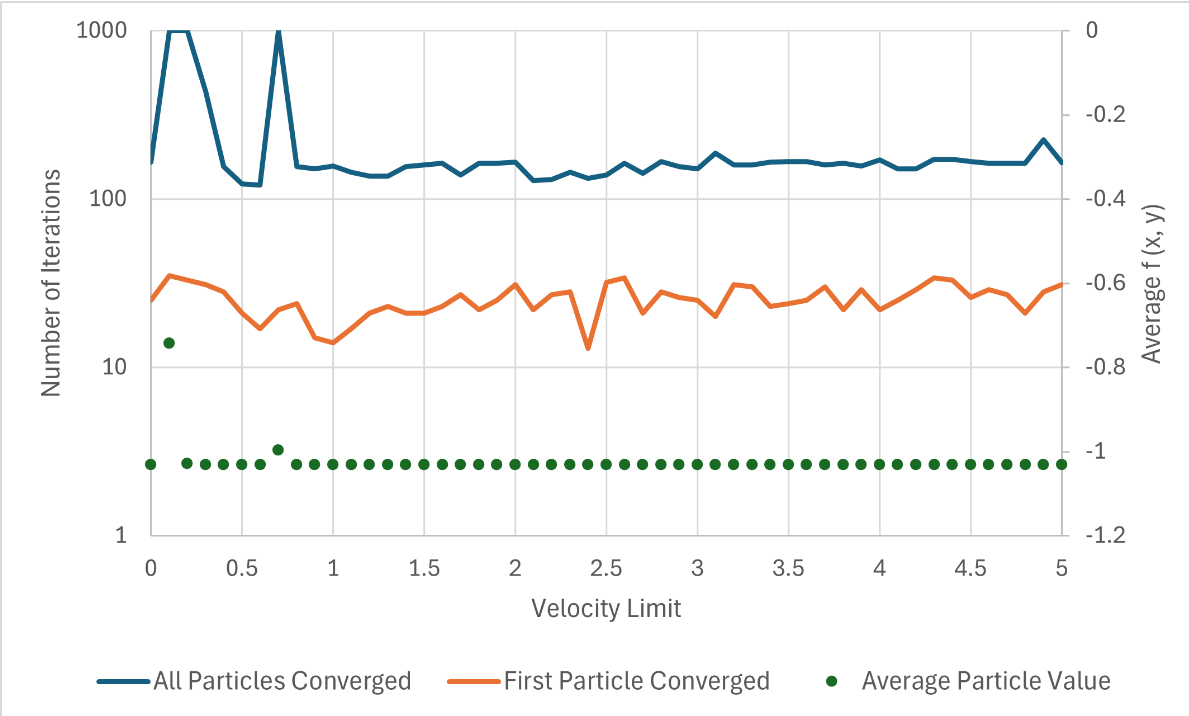

Figure 2: Plot demonstrating the effect of the velocity limit on the convergence speed of the first and all particles and the average fitness value of all particles at the end of the simulation.

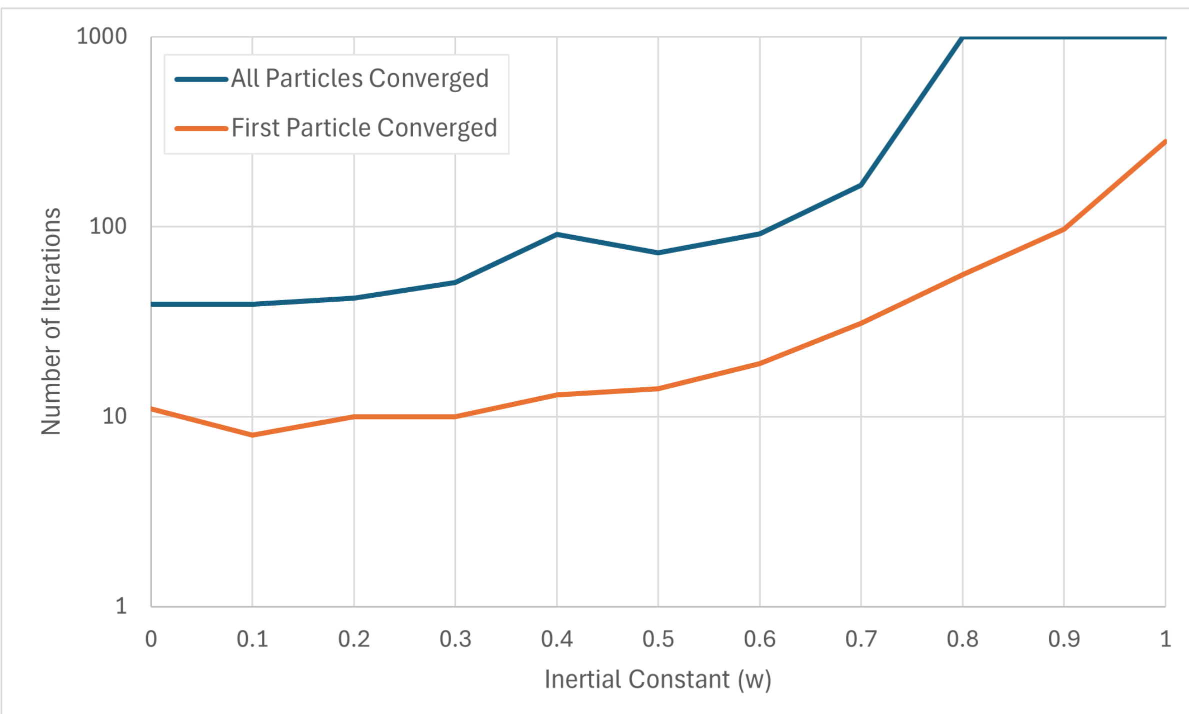

Figure 3: Plot demonstrating the effect of the inertia weight on the convergence speed of the first and all particles.

3.4 Effect of Acceleration Constants

The swarm tripped the patience threshold and failed to converge when the social acceleration constant c_2 was set to zero. This is demonstrated by the green dots in Figure 4. In this scenario, the particles seemed to quickly settle into whatever local minimum was closest and then stay there because there was no other reference point to urge them out. Furthermore, Figure 4 demonstrates that the ratio of cognitive and social accelerations does not have a strong influence on how quickly the first particle finds the global minimum. However, as the cognitive acceleration c_1 increasingly dominates the social acceleration c_2, it takes longer for the remainder of the particles to reach the global minimum. This is expected since a larger c_1 implies that the particles are more inclined to explore their P_{best} than the global G_best.

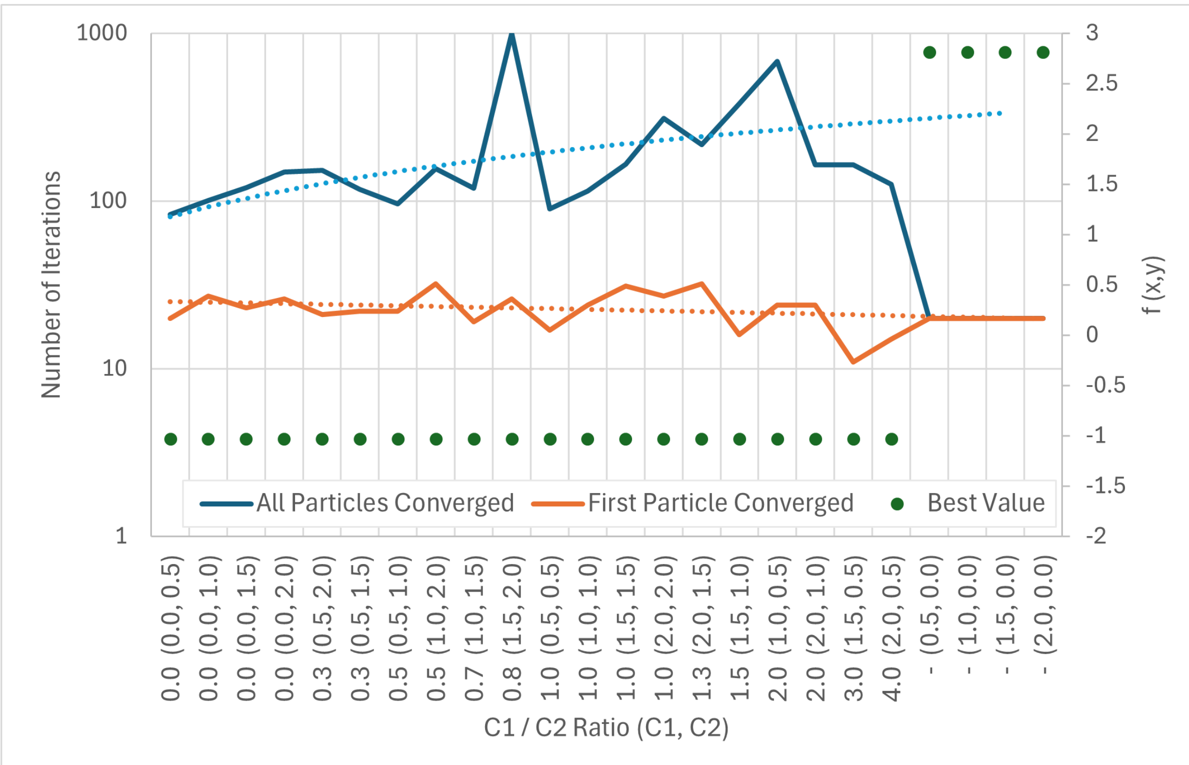

Figure 4: Plot demonstrating the effect of the acceleration constants c_1 and c_2 on the convergence speed of the first and all particles and the best fitness value of all particles at the end of the simulation.

4 Conclusion

This study implemented and analyzed a Particle Swarm Optimization (PSO) algorithm to minimize the six-hump camelback function. The implementation utilized a modular design with a *particle* class to represent individual particles and a *swarm* class to manage the collective behavior of the swarm. The PSO algorithm was evaluated using a variety of parameter configurations to understand their effects on convergence speed, solution quality, and swarm dynamics.

The baseline experiment demonstrated the algorithm's capability to achieve the global minimum of $z = -1.03163$ with 25 particles, velocity limit of 2.0, $w = 0.7$, and $c_1 = c_2 = 1.5$. This configuration required 31 iterations for the first particle to find the global minimum and 166 iterations for all particles to converge.

The parameter studies highlighted several key findings:

- Increasing the particle count improved the convergence speed of the first particle to the global minimum but had minimal effect on the time required for all particles to converge.
- Velocity limits influenced convergence behavior by balancing exploration and exploitation. Low velocity limits restricted distant particles from reaching the global minimum within the maximum iterations. Excessively high limits should have produced overshooting and instability, but this was not observed likely due to the stability of the other parameters.
- The inertia weight, w, played a critical role in swarm dynamics. Moderate values of w in the range $[0.1, 0.3]$ yielded the fastest convergence rates, whereas higher values caused overshooting, preventing some particles from converging.
- The balance between cognitive (c_1) and social (c_2) acceleration constants affected swarm behavior. When $c_2 = 0$, particles settled in local minima due to a lack of social influence, highlighting the importance of the global best position. Additionally, higher c_1 values prolonged the convergence of all particles, emphasizing the need for a balanced ratio between exploration and collaboration.

Overall, the experiments demonstrated that careful tuning of PSO parameters is essential to achieve optimal performance. However, the experience of tuning the PSO parameters seemed less extreme when compared to the tuning of parameters of other computational intelligence methods. The results provide valuable insights into the relationships between parameter settings and convergence dynamics.

References

[1] W. Qiao, *Particle swarm optimization (pso)*, ECEN-935: Computational Intelligence, October 2024, 2024.

5 Appendix

5.1 main.m

```
%{
    File: main.m
    Author: Zachary M Swanson
    Date: 11-20-2024
    Description: This file contains the main script which is used to run the
    particle swarm optimization algorithm on the given fitness function. The
    script will run the algorithm with the given parameters and save the results
    to a file. It will also run an ablation study on the parameters to determine
    the effect of each parameter on the performance of the algorithm.
%}

% Define the six-hump camel back function as the fitness function
function z = fitness_func(x, y)
    z = (4 - 2.1 * x.^2 + x.^4/3) .* x.^2 + x.*y + (-4 + 4 * y.^2) .* y.^2;
end

min_x = -5;
max_x = 5;
min_y = -5;
max_y = 5;

min_threshold = 1e-6;
patience = 20;
max_iterations = 1000;

num_particles = 25;
v_max = 0.2 * (max_x - min_x);
accel_c1 = 1.5;
accel_c2 = 1.5;
inertia_w = 0.7;

rng(52);
swarm_obj = swarm(...
    num_particles, min_threshold, max_iterations, patience, accel_c1, ...
    accel_c2, inertia_w, v_max, max_x, max_y, min_x, min_y, @fitness_func);

[x, y, z, avg_z, max_z, iters] = swarm_obj.run();
first_min_iters = swarm_obj.iters_to_first_min();
fprintf('\nBest␣x:␣%f\n', x);
fprintf('Best␣y:␣%f\n', y);
fprintf('Best␣z:␣%f\n', z);
fprintf('Iterations:␣%d\n', iters);

% Write the results to a file using the following format:
% best_x,best_y,best_z,avg_z,max_z,iters,first_min_iters,n_particle,v_max, ...
% ... inertia_w,accel_c1,accel_c2
fileID = fopen('results.csv', 'a');
fprintf(fileID, '%f,%f,%f,%f,%f,%d,%d,%d,%f,%f,%f,%f\n', ...
    x, y, z, avg_z, max_z, iters, first_min_iters, num_particles, v_max, ...
    inertia_w, accel_c1, accel_c2);
fclose(fileID);

% Save the history to a file
filename = sprintf('histories/swarm_obj_%d_%.2f_%.2f_%.2f_%.2f.mat', ...
    num_particles, v_max, inertia_w, accel_c1, accel_c2);
save(filename, 'swarm_obj');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%     Ablation Study     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%

range_n_particles = 1:1:50;
range_v_max = (0:0.01:0.5) * (max_x - min_x);
range_inertia_w = 0:0.1:1.0;
```

```
range_accel_c1 = 0:0.5:2.0;
range_accel_c2 = 0:0.5:2.0;

disp('Running ablation study...');
disp('Starting number of particles study...');

for n = range_n_particles
    rng(52);
    swarm_obj = swarm(...
        n, min_threshold, max_iterations, patience, accel_c1, ...
        accel_c2, inertia_w, v_max, max_x, max_y, min_x, min_y, @fitness_func);

    [x, y, z, avg_z, max_z, iters] = swarm_obj.run();
    first_min_iters = swarm_obj.iters_to_first_min();

    fileID = fopen('results.csv', 'a');
    fprintf(fileID, '%f,%f,%f,%f,%f,%d,%d,%d,%f,%f,%f,%f\n', ...
        x, y, z, avg_z, max_z, iters, first_min_iters, n, v_max, ...
        inertia_w, accel_c1, accel_c2);
    fclose(fileID);


    filename = sprintf('histories/swarm_obj_%d_%.2f_%.2f_%.2f_%.2f.mat', ...
        n, v_max, inertia_w, accel_c1, accel_c2);
    save(filename, 'swarm_obj');
end

disp('Starting velocity limit study...');

for v = range_v_max
    rng(52);
    swarm_obj = swarm(...
        num_particles, min_threshold, max_iterations, patience, accel_c1, ...
        accel_c2, inertia_w, v, max_x, max_y, min_x, min_y, @fitness_func);

    [x, y, z, avg_z, max_z, iters] = swarm_obj.run();
    first_min_iters = swarm_obj.iters_to_first_min();

    fileID = fopen('results.csv', 'a');
    fprintf(fileID, '%f,%f,%f,%f,%f,%d,%d,%d,%f,%f,%f,%f\n', ...
        x, y, z, avg_z, max_z, iters, first_min_iters, num_particles, v, ...
        inertia_w, accel_c1, accel_c2);
    fclose(fileID);

    filename = sprintf('histories/swarm_obj_%d_%.2f_%.2f_%.2f_%.2f.mat', ...
        num_particles, v, inertia_w, accel_c1, accel_c2);
    save(filename, 'swarm_obj');
end

disp('Starting inertia_w study...');

for w = range_inertia_w
    rng(52);
    swarm_obj = swarm(...
        num_particles, min_threshold, max_iterations, patience, accel_c1, ...
        accel_c2, w, v_max, max_x, max_y, min_x, min_y, @fitness_func);

    [x, y, z, avg_z, max_z, iters] = swarm_obj.run();
    first_min_iters = swarm_obj.iters_to_first_min();

    fileID = fopen('results.csv', 'a');
    fprintf(fileID, '%f,%f,%f,%f,%f,%d,%d,%d,%f,%f,%f,%f\n', ...
        x, y, z, avg_z, max_z, iters, first_min_iters, num_particles, ...
        v_max, w, accel_c1, accel_c2);
    fclose(fileID);

    filename = sprintf('histories/swarm_obj_%d_%.2f_%.2f_%.2f_%.2f.mat', ...
        num_particles, v_max, w, accel_c1, accel_c2);
```

```
    save(filename, 'swarm_obj');
end

disp('Starting accel_c1 and accel_c2 study...');

for c1 = range_accel_c1
    for c2 = range_accel_c2
        if c1 == 0 && c2 == 0
            continue;
        end

        rng(52);
        swarm_obj = swarm(...
            num_particles, min_threshold, max_iterations, patience, c1, ...
            c2, inertia_w, v_max, max_x, max_y, min_x, min_y, @fitness_func);

        [x, y, z, avg_z, max_z, iters] = swarm_obj.run();
        first_min_iters = swarm_obj.iters_to_first_min();

        fileID = fopen('results.csv', 'a');
        fprintf(fileID, '%f,%f,%f,%f,%f,%d,%d,%d,%f,%f,%f,%f\n', ...
            x, y, z, avg_z, max_z, iters, first_min_iters, ...
            num_particles, v_max, inertia_w, c1, c2);
        fclose(fileID);

        filename = sprintf('histories/swarm_obj_%d_%.2f_%.2f_%.2f_%.2f.mat', ...
            num_particles, v_max, inertia_w, c1, c2);
        save(filename, 'swarm_obj');
    end
end
```

5.2 swarm.m

```
%{
    File: swarm.m
    Author: Zachary M Swanson
    Date: 11-20-2024
    Description: This file contains the swarm class which is used to implement
    the particle swarm optimization algorithm. The swarm class is used to
    initialize a swarm of particles and run the optimization algorithm to find
    the minimum value of a given fitness function.
%}

classdef swarm < handle
    properties
        num_particles
        particles
        gbest_x
        gbest_y
        gbest_val
        threshold
        num_iterations
        patience
        accel_c1
        accel_c2
        inertia_w
        max_vel
        max_x
        max_y
        min_x
        min_y
        fitness_func
        history
    end

    methods
        function obj = swarm(num_particles, threshold, num_iters, patience, ...
                accel_c1, accel_c2, inertia_w, max_vel, max_x, max_y, ...
                min_x, min_y, fitness_func)
            obj.num_particles = num_particles;
            obj.threshold = threshold;
            obj.num_iterations = num_iters;
            obj.patience = patience;
            obj.accel_c1 = accel_c1;
            obj.accel_c2 = accel_c2;
            obj.inertia_w = inertia_w;
            obj.max_vel = max_vel;
            obj.max_x = max_x;
            obj.max_y = max_y;
            obj.min_x = min_x;
            obj.min_y = min_y;
            obj.fitness_func = fitness_func;

            obj.history = zeros(num_iters, num_particles, 3);

            obj.particles = particle.empty(obj.num_particles, 0);
            for i = 1:obj.num_particles
                % Randomly initialize the position and velocity of the particle
                % within the bounds of the search space
                pos_x = rand() * (obj.max_x - obj.min_x) + obj.min_x;
                pos_y = rand() * (obj.max_y - obj.min_y) + obj.min_y;
                vel_x = 0;
                vel_y = 0;
                obj.particles(i) = particle(pos_x, pos_y, vel_x, vel_y, ...
                    obj.fitness_func, obj.threshold);

                obj.history(1, i, 1) = pos_x;
                obj.history(1, i, 2) = pos_y;
```

```
                obj.history(1, i, 3) = obj.particles(i).crnt_val;

                % Update the global best position based on the initial positions
                if i == 1
                    obj.gbest_x = obj.particles(i).pos_x;
                    obj.gbest_y = obj.particles(i).pos_y;
                    obj.gbest_val = obj.particles(i).best_val;
                else
                    if obj.particles(i).best_val < obj.gbest_val
                        obj.gbest_x = obj.particles(i).pos_x;
                        obj.gbest_y = obj.particles(i).pos_y;
                        obj.gbest_val = obj.particles(i).best_val;
                    end
                end
            end
        end

        function [x, y, val, avg_z, max_z, iters] = run(obj)
            iters = 0;

            for i = 1:obj.num_iterations
                no_progress = true;

                % Loop through each particle and update its position
                for j = 1:obj.num_particles
                    obj.particles(j).update(obj.accel_c1, ...
                        obj.accel_c2, obj.inertia_w, obj.gbest_x, ...
                        obj.gbest_y, obj.max_x, obj.max_y, obj.min_x, ...
                        obj.min_y, obj.max_vel);

                    % Update the current value of the particle in the history
                    % matrix for later analysis
                    obj.history(i, j, 1) = obj.particles(j).pos_x;
                    obj.history(i, j, 2) = obj.particles(j).pos_y;
                    obj.history(i, j, 3) = obj.particles(j).crnt_val;
                end

                % Loop through each particle and update the global best position
                for j = 1:obj.num_particles
                    if obj.particles(j).best_val < obj.gbest_val
                        obj.gbest_x = obj.particles(j).pos_x;
                        obj.gbest_y = obj.particles(j).pos_y;
                        obj.gbest_val = obj.particles(j).best_val;
                    end

                    % Check if the particle has made progress within the
                    % last patience iterations
                    if obj.particles(j).no_progress_count < obj.patience
                        no_progress = false;
                    end
                end

                % if no particle has made progress within the last patience
                % iterations, break out of the loop... we're assuming that
                % all particles have converged to some minimum
                if no_progress
                    iters = i;
                    break;
                end
            end

            x = obj.gbest_x;
            y = obj.gbest_y;
            val = obj.gbest_val;

            if iters == 0
                iters = obj.num_iterations;
            end
```

```matlab

            avg_z = 0;
            max_z = 0;

            for i = 1:obj.num_particles
                avg_z = avg_z + obj.particles(i).crnt_val;
                if i == 1
                    max_z = obj.particles(i).crnt_val;
                else
                    if obj.particles(i).crnt_val > max_z
                        max_z = obj.particles(i).crnt_val;
                    end
                end
            end

            avg_z = avg_z / obj.num_particles;
        end

        % Helper function to find the iteration at which the first minimum
        % value was found based on the history of the swarm
        function iters = iters_to_first_min(obj)
            % find the minimum value in the history
            min_val = obj.history(1, 1, 3);
            min_i = 0;

            for i = 1:obj.num_iterations
                for j = 1:obj.num_particles
                    if obj.history(i, j, 3) < min_val
                        min_val = obj.history(i, j, 3);
                    end
                end
            end

            if min_val < 0
                min_val = 0.9999 * min_val;
            elseif min_val > 0
                min_val = 1.0001 * min_val;
            end

            iter_found = false;

            for i = 1:obj.num_iterations
                for j = 1:obj.num_particles
                    if obj.history(i, j, 3) <= min_val
                        min_i = i;
                        iter_found = true;
                        break;
                    end
                end

                if iter_found
                    break;
                end
            end

            iters = min_i;
        end
    end
end
```

5.3 particle.m

```
%{
    File: particle.m
    Author: Zachary M Swanson
    Date: 11-20-2024
    Description: This file contains the particle class which is used to represent
    a particle in the particle swarm optimization algorithm. The particle class
    contains properties for the position, velocity, fitness function, and best
    position of the particle. It also contains methods to update the particle's
    position and velocity based on the particle swarm optimization algorithm.
%}

classdef particle < handle
    properties
        pos_x
        pos_y
        vel_x
        vel_y
        fitness_func
        best_pos_x
        best_pos_y
        best_val
        crnt_val
        no_progress_count
        threshold
    end

    methods
        function obj = particle(pos_x, pos_y, vel_x, vel_y, fit_fn, threshold)
            obj.pos_x = pos_x;
            obj.pos_y = pos_y;
            obj.vel_x = vel_x;
            obj.vel_y = vel_y;

            obj.best_pos_x = pos_x;
            obj.best_pos_y = pos_y;

            % Evaluate the fitness function at the initial position to get the
            % initial p_best value
            obj.fitness_func = fit_fn;
            obj.crnt_val = obj.fitness_func(obj.pos_x, obj.pos_y);
            obj.best_val = obj.crnt_val;

            obj.no_progress_count = 0;
            obj.threshold = threshold;
        end

        % Update the particle's position and velocity based on the PSO algorithm
        function [x, y] = update(obj, c1, c2, w, gbest_x, gbest_y, ...
                max_x, max_y, min_x, min_y, max_vel)
            phi_1 = rand();
            phi_2 = rand();

            inertia_x = w * obj.vel_x;
            inertia_y = w * obj.vel_y;

            cognitive_x = c1 * phi_1 * (obj.best_pos_x - obj.pos_x);
            cognitive_y = c1 * phi_1 * (obj.best_pos_y - obj.pos_y);

            social_x = c2 * phi_2 * (gbest_x - obj.pos_x);
            social_y = c2 * phi_2 * (gbest_y - obj.pos_y);

            % Combine the three components to get the new velocity
            obj.vel_x = inertia_x + cognitive_x + social_x;
            obj.vel_y = inertia_y + cognitive_y + social_y;

```

```
            % Limit the velocity if necessary
            if max_vel > 0
                if obj.vel_x > max_vel
                    obj.vel_x = max_vel;
                elseif obj.vel_x < -max_vel
                    obj.vel_x = -max_vel;
                end

                if obj.vel_y > max_vel
                    obj.vel_y = max_vel;
                elseif obj.vel_y < -max_vel
                    obj.vel_y = -max_vel;
                end
            end

            % Update the particle's position
            obj.pos_x = obj.pos_x + obj.vel_x;
            obj.pos_y = obj.pos_y + obj.vel_y;

            % Ensure the particle stays within the search space
            if obj.pos_x > max_x
                obj.pos_x = max_x;
            elseif obj.pos_x < min_x
                obj.pos_x = min_x;
            end

            if obj.pos_y > max_y
                obj.pos_y = max_y;
            elseif obj.pos_y < min_y
                obj.pos_y = min_y;
            end

            % Evaluate the new position with the fitness function
            val = obj.fitness_func(obj.pos_x, obj.pos_y);

            % Update the best position if necessary
            if val < obj.best_val
                obj.best_val = val;
                obj.best_pos_x = obj.pos_x;
                obj.best_pos_y = obj.pos_y;
            end

            % Check for progress
            if abs(val - obj.crnt_val) < obj.threshold
                obj.no_progress_count = obj.no_progress_count + 1;
            else
                obj.no_progress_count = 0;
            end

            obj.crnt_val = val;

            x = obj.pos_x;
            y = obj.pos_y;
        end
    end
end
```