Assignment 4 Solution

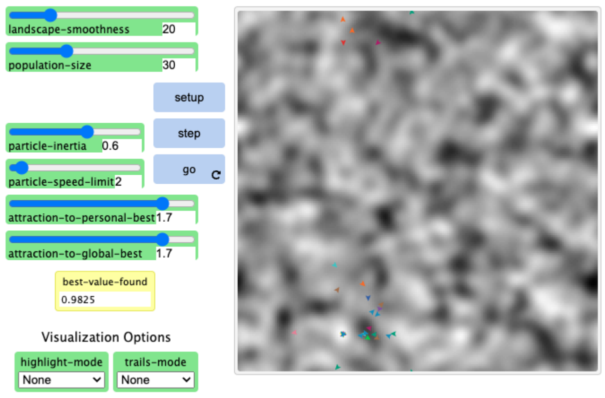
Jacob Bodera

2024-12-01

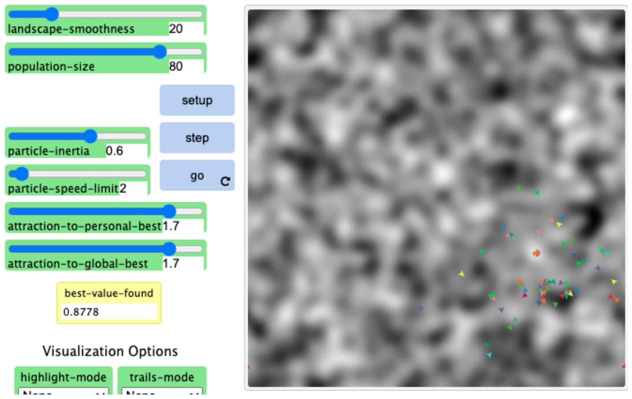
Problem 1 Solution

Experimentation with PSO in NetLogo

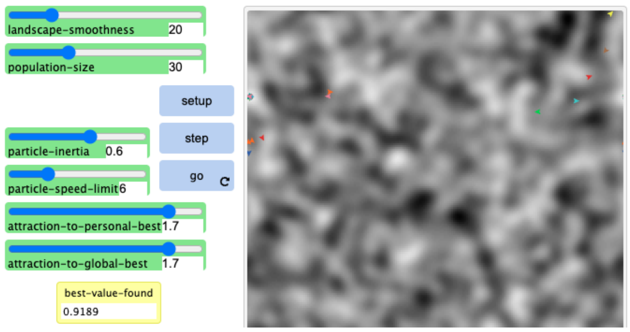
I ran seven different experiments in the NetLogo simulation to see how different parameters affect the swarms ability to location the global minimum of the search space. The parameters that were tested include the population size (p), speed limit (s), particle inertia (i), personal-best factor (pb), and global-best factor (gb). The plots below show some of the results from experimenting with this model.



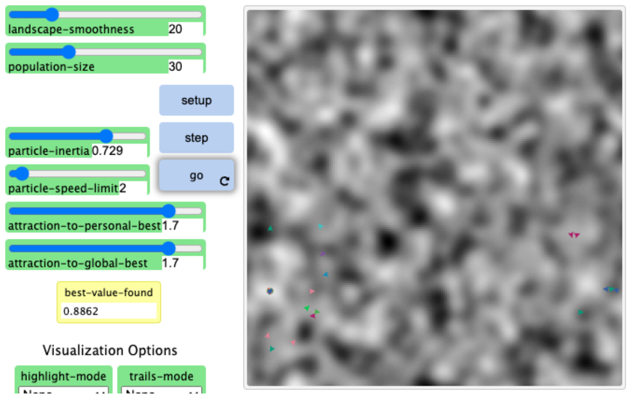
landscape-smoothness 20
population-size 30
particle-inertia 0.6
particle-speed-limit 2
attraction-to-personal-best 1.7
attraction-to-global-best 1.7
best-value-found 0.9825
Visualization Options
highlight-mode None
trails-mode None



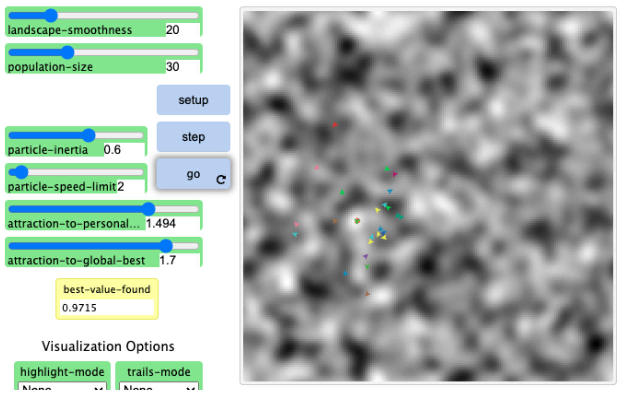
landscape-smoothness 20
population-size 80
particle-inertia 0.6
particle-speed-limit 2
attraction-to-personal-best 1.7
attraction-to-global-best 1.7
best-value-found 0.8778
Visualization Options
highlight-mode
trails-mode



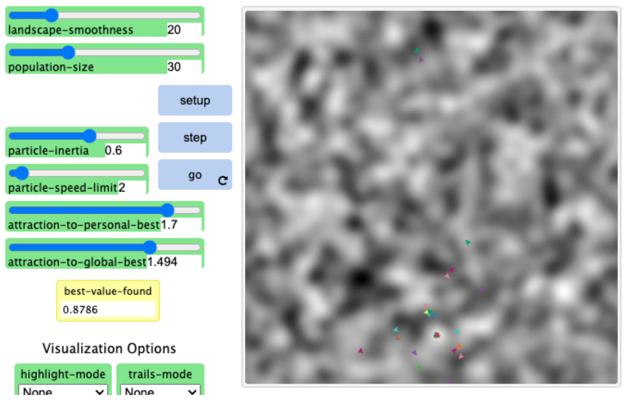
landscape-smoothness 20
population-size 30
particle-inertia 0.6
particle-speed-limit 6
attraction-to-personal-best 1.7
attraction-to-global-best 1.7
best-value-found 0.9189



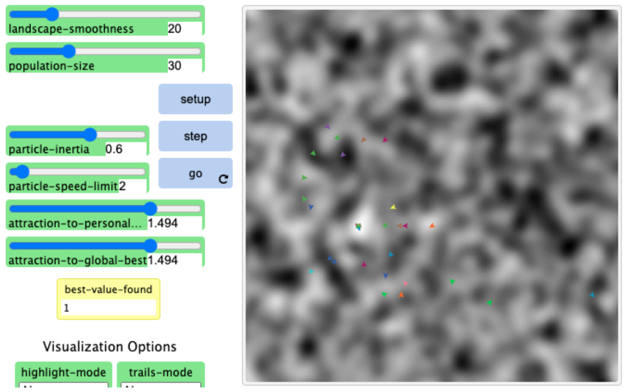
landscape-smoothness 20
population-size 30
particle-inertia 0.729
particle-speed-limit 2
attraction-to-personal-best 1.7
attraction-to-global-best 1.7
best-value-found 0.8862
Visualization Options
highlight-mode
trails-mode



landscape-smoothness 20
population-size 30
particle-inertia 0.6
particle-speed-limit 2
attraction-to-personal... 1.494
attraction-to-global-best 1.7
best-value-found 0.9715
Visualization Options
highlight-mode
trails-mode



landscape-smoothness 20
population-size 30
particle-inertia 0.6
particle-speed-limit 2
attraction-to-personal-best 1.7
attraction-to-global-best 1.494
best-value-found 0.8786
Visualization Options
highlight-mode None
trails-mode None


landscape-smoothness 20

population-size 30

setup

step

go

particle-inertia 0.6

particle-speed-limit 2

attraction-to-personal... 1.494

attraction-to-global-best 1.494

best-value-found

1

Visualization Options

highlight-mode

trails-mode

Problem 2 Solution

Problem Formulation

State Representation: For a swarm of size n, the state representation of this PSO algorithm is a n by 2 matrix corresponding to the position and velocity of each particle in the swarm.

$$state = \begin{bmatrix} p_1 & v_1 \\ p_2 & v_2 \\ .. & .. \\ p_n & v_n \end{bmatrix}$$

Initial State: The initial state was chosen to be a set of particles with randomly initialized positions within the bounds of the search space and with zero initial velocities.

Goal State: The goal state is chosen to be the state in which at least one of the particles in the swarm minimized the six-hump camelback function. This is of course with some acceptance of tolerance, chosen arbitrarily to be 0.1% error.

Actions: There are two main actions that can be perform on each state: updating the velocity of each particle and updating the position of each particle. These updating equations are generally expressed as the following:

$$v_{i+1} = wv_i + c_1 r_1 (P_{best} - x_i) + c_2 r_2 (N_{best} - x_i)$$

$$x_{i+1} = x_i + v_{i+1}$$

where w is the inertia of the particle, c_1 and c_2 are acceleration factors, r_1 and r_2 are randomly generated numbers, and i is the iteration number.

Cost: The cost for this algorithm is the six hump camel back function described as:

$$z = (4 - 2.1x^2 + \frac{x^4}{3})x^2 + xy + (-4 + 4y^2)y^2$$

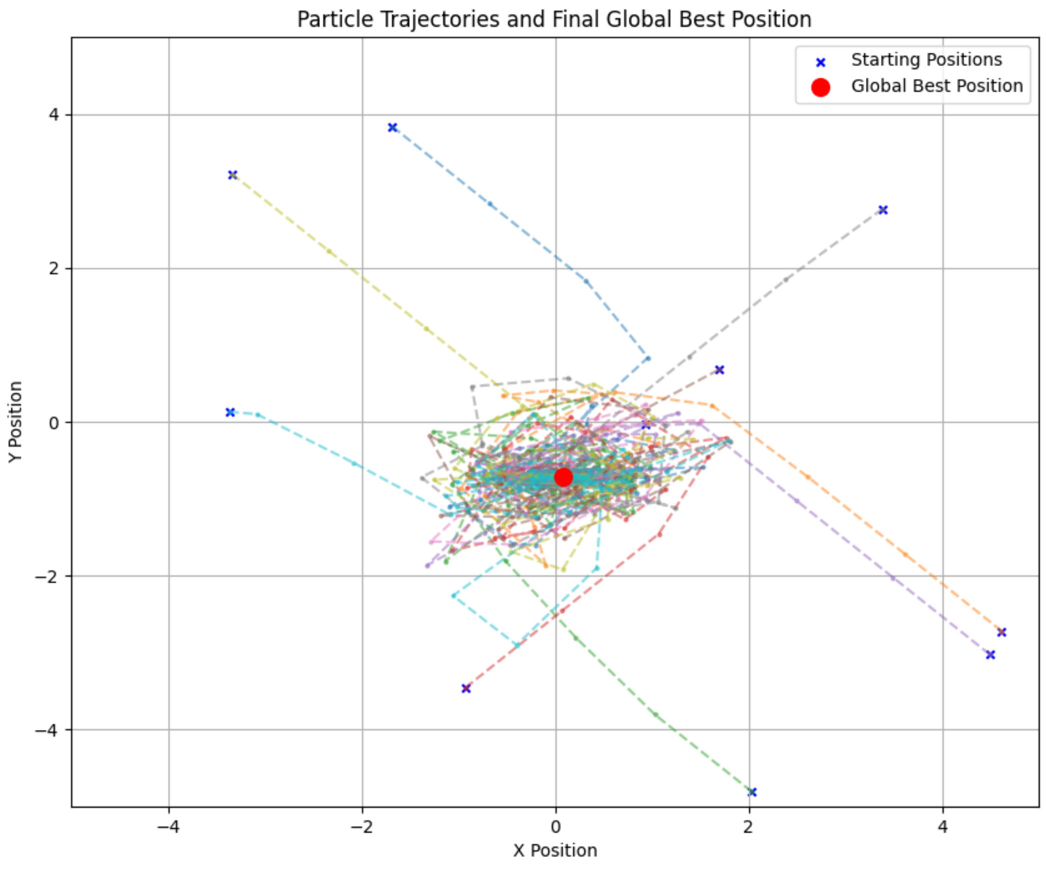
Experimenting with Different Velocity Updating Equations

Interia Weight Updating

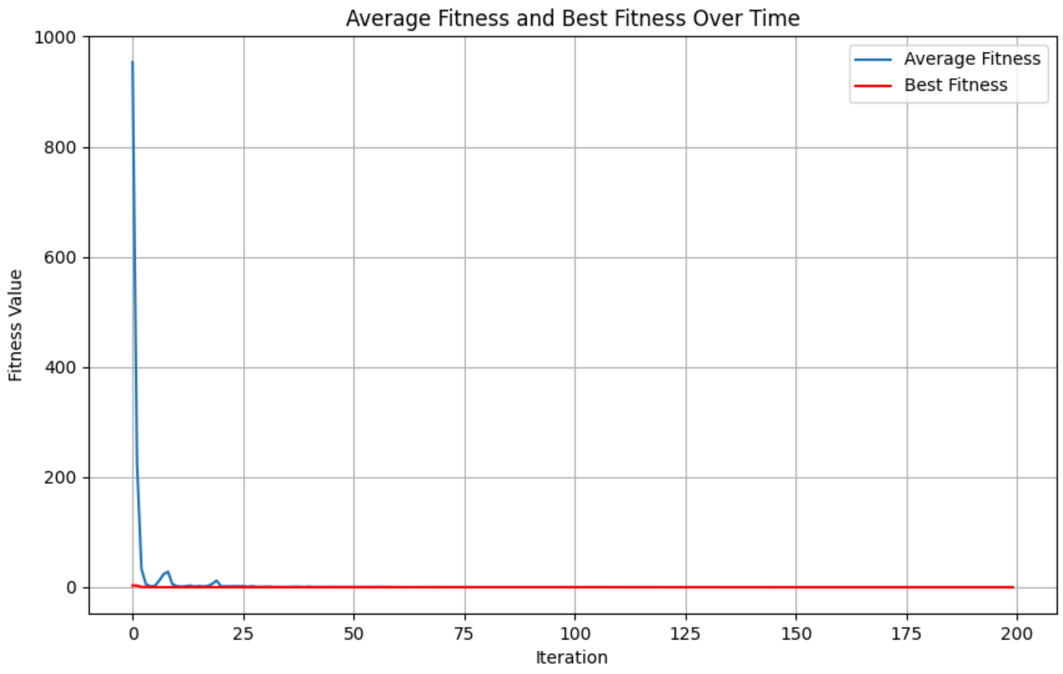
Best Position: [0.08981815 -0.71265339]

Best Fitness: -1.031628453434532

% Error: 2.0270804646806416e-09



Particle Trajectories and Final Global Best Position
Starting Positions
Global Best Position
X Position
Y Position
−4
−2
0
2
4

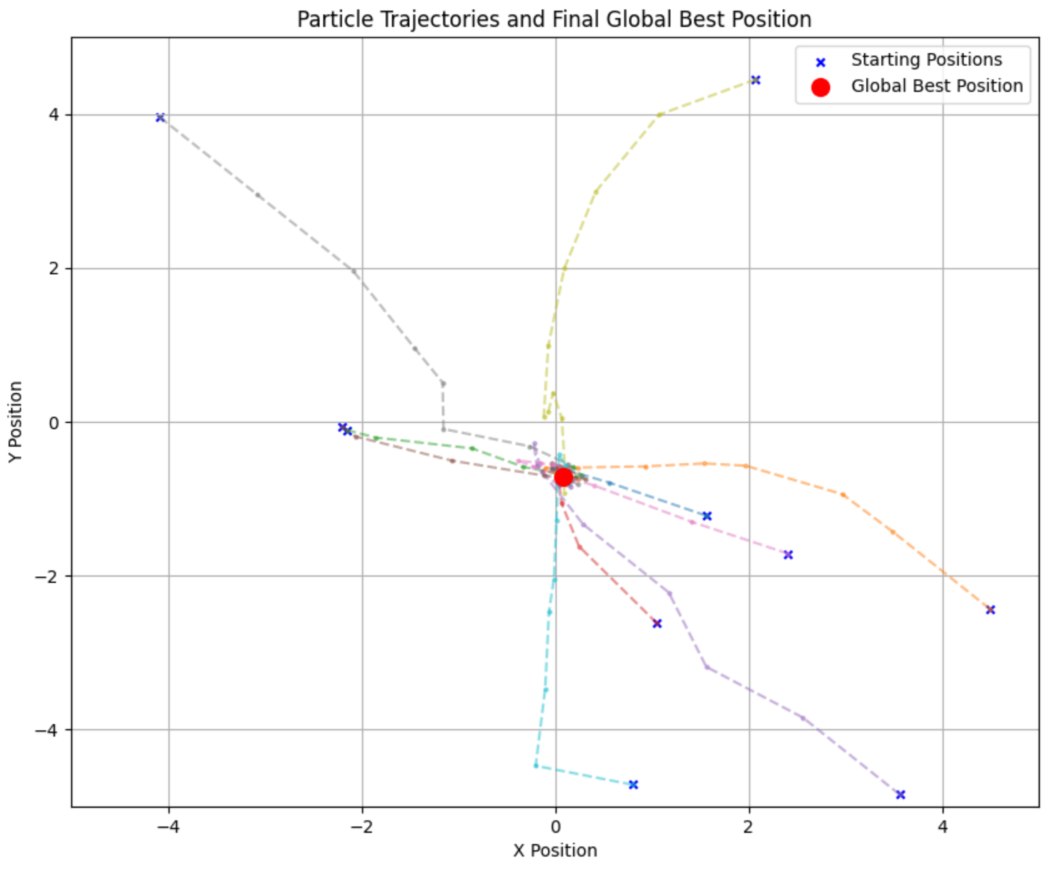


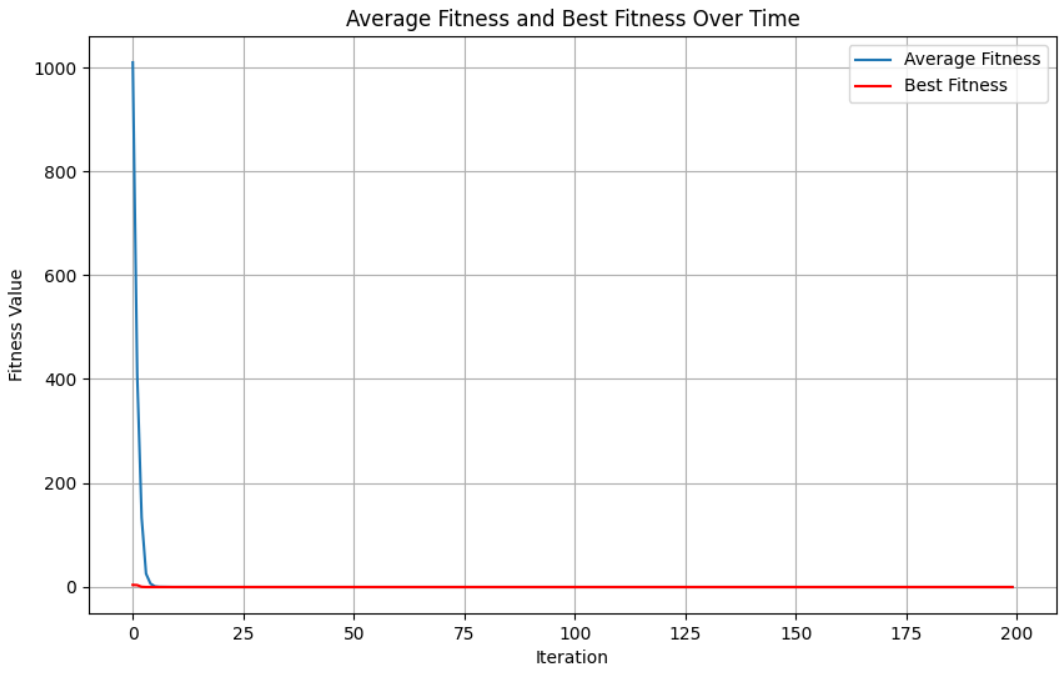
Constriction Factor Updating

Best Position: [0.08984201 -0.7126564]

Best Fitness: -1.0316284534898774

% Error: 7.391948579053364e-09


Particle Trajectories and Final Global Best Position
Starting Positions
Global Best Position
X Position
Y Position
−4
−2
0
2
4
4
2
0
−2
−4

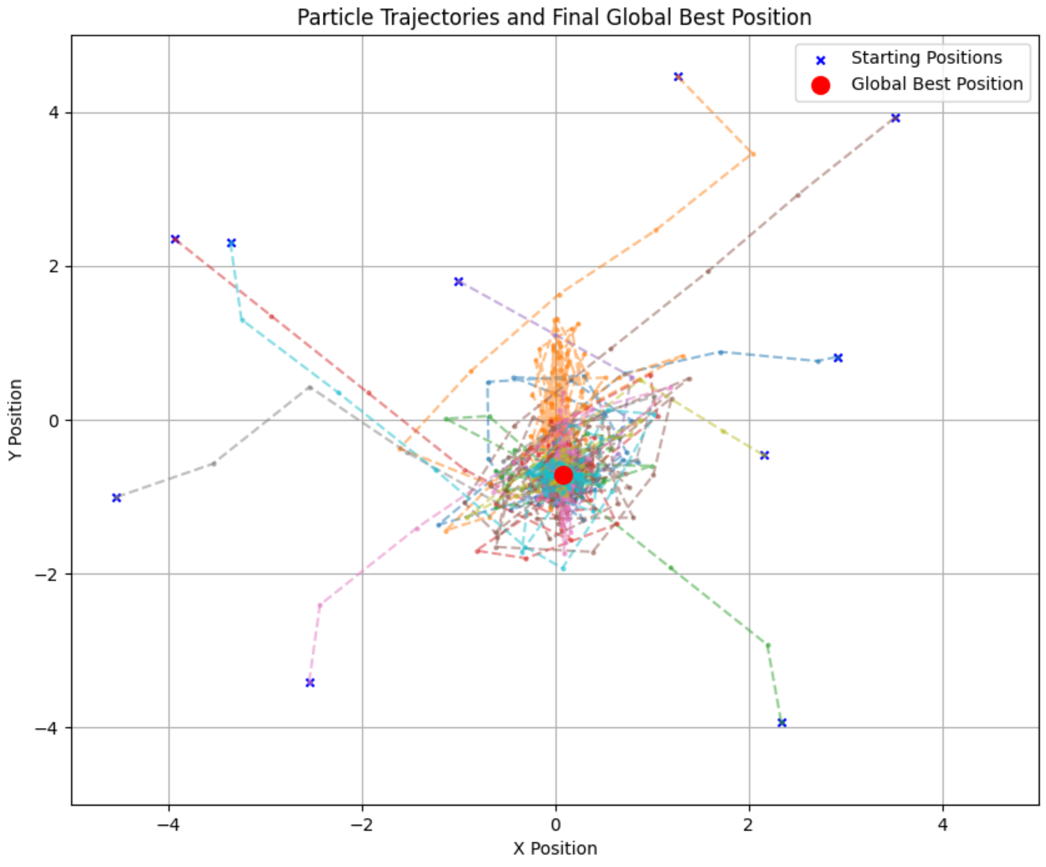


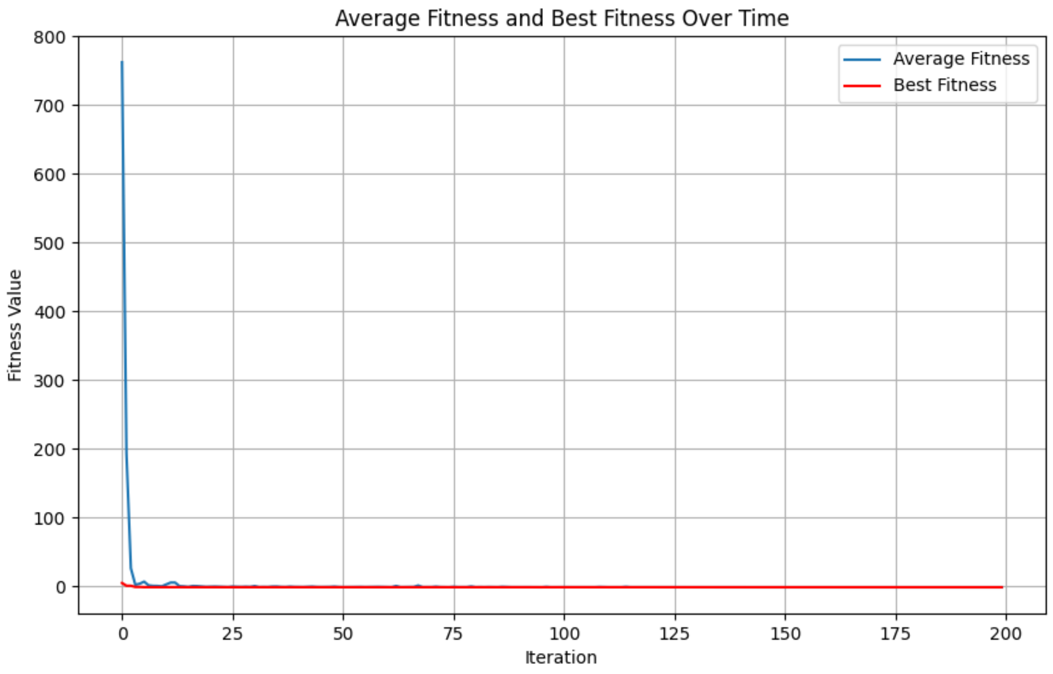
GCPSO Updating

Best Position: [0.08984095 -0.71272291]

Best Fitness: -1.0316284534512004

% Error: 3.642821514411966e-09


Particle Trajectories and Final Global Best Position
Starting Positions
Global Best Position
Y Position
X Position
4
2
0
−2
−4
−4
−2
0
2
4


Average Fitness and Best Fitness Over Time
Average Fitness
Best Fitness
Fitness Value
Iteration
800
700
600
500
400
300
200
100
0
0
25
50
75
100
125
150
175
200

Problem 3 Solution

Appendix A: Source Code

Question 3

```
import random

PROB_MUTATION_VERSUS_CROSS = 0.4
PROB_MUTATION = 0.1

terminals = ['a0', 'a1', 'd0', 'd1', 'd2', 'd3']
functions = {
    'AND': lambda x, y: 0 if x == 0 else y,
    'OR': lambda x, y: 1 if x == 1 else y,
    'NOT': lambda x: not x,
    'IF': lambda x, y, z: y if x else z
}

def random_terminal():
    return random.choice(terminals)

def random_function():
    return random.choice(list(functions.keys()))

def fitness(program):
    num_correct = 0
    for a0 in [0, 1]:
        for a1 in [0, 1]:
            for d0 in [0, 1]:
                for d1 in [0, 1]:
                    for d2 in [0, 1]:
                        for d3 in [0, 1]:
                            inputs = {'a0': a0, 'a1': a1, 'd0': d0, 'd1': d1, 'd2': d2, 'd3': d3}
                            expected_d = inputs[f'd{(a0 * 2) + a1}']
                            if evaluate_program(program, inputs) == expected_d:
                                num_correct += 1
    return num_correct / 64.0

def generate_program(depth=3):
    if depth == 0 or (depth > 1 and random.random() < 0.5):
        return random_terminal()
    else:
        function = random_function()
        if function == 'NOT':
            return [function, generate_program(depth - 1)]
        elif function == 'IF':
            return [function, generate_program(depth - 1), generate_program(depth - 1), generate_progra
        else:
            return [function, generate_program(depth - 1), generate_program(depth - 1)]

def evaluate_program(program, inputs):
    if isinstance(program, str):
        return inputs[program]
    function = program[0]
    if function == 'NOT':
        return functions[function](evaluate_program(program[1], inputs))
    elif function == 'IF':
        return functions[function](evaluate_program(program[1], inputs),
                                   evaluate_program(program[2], inputs),
                                   evaluate_program(program[3], inputs))
    else:
        return functions[function](evaluate_program(program[1], inputs),
                                   evaluate_program(program[2], inputs))

def mutate(program, depth=3):
    if random.random() < PROB_MUTATION:
        return generate_program(depth)
    if isinstance(program, list):
        if program[0] == 'NOT':
            return [program[0],
                    mutate(program[1], depth - 1)]
        elif program[0] == 'IF':
            return [program[0],
                    mutate(program[1], depth - 1),
                    mutate(program[2], depth - 1),
                    mutate(program[3], depth - 1)]
        else:
            return [program[0],
                    mutate(program[1], depth - 1),
                    mutate(program[2], depth - 1)]
    return program

def crossover(parent1, parent2):
    if isinstance(parent1, str) or isinstance(parent2, str):
        return parent2 if random.random() < 0.5 else parent1
    if len(parent1) != len(parent2):
        return parent1
    return [parent1[0]] + [crossover(p1, p2) for p1, p2 in zip(parent1[1:], parent2[1:])]

def start_environment(population_size, generations):
    population = [generate_program() for _ in range(population_size)]
    best_fitness = 0
    for g in range(generations):
        population = sorted(population, key=lambda p: fitness(p), reverse=True)
        best_fitness = fitness(population[0])
        print(f"Generation: {g} --- Best Fitness: {best_fitness}")

        if best_fitness == 1.0:
            return population[0], best_fitness

        new_population = population[:10]
        while len(new_population) < population_size:
            if random.random() > PROB_MUTATION_VERSUS_CROSS:
                parent1, parent2 = random.choices(population[:population_size // 2], k=2)
                p_new = crossover(parent1, parent2)
            else:
                p = random.choice(population[:population_size // 2])
                p_new = mutate(p)
            new_population.append(p_new)
        population = new_population

    return population[0], best_fitness

solution, fitness = start_environment(500, 500)

print(solution)
print(fitness)
```