Assignment 4

Developing Particle Swarm Optimization (PSO) in C with Case Study

Furqaan Khurram Qamar

400514719

1 Introduction

In this assignment, I developed an optimization solution utilizing the Particle Swarm Optimization method. This was tested through minimizing the following functions:

Griewank Function

$$f(\mathbf{x}) = 1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right)$$

Lévy Function

$$f(\mathbf{x}) = \sin^2(\pi w_1) + \sum_{i=1}^{n-1}(w_i - 1)^2\left[1 + 10\sin^2(\pi w_i + 1)\right] + (w_n - 1)^2\left[1 + \sin^2(2\pi w_n)\right]$$

where $w_i = 1 + \frac{x_i - 1}{4}$

Rastrigin Function

$$f(\mathbf{x}) = 10n + \sum_{i=1}^{n}\left[x_i^2 - 10\cos(2\pi x_i)\right]$$

Rosenbrock Function

$$f(\mathbf{x}) = \sum_{i=1}^{n-1}\left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$$

Schwefel Function

$$f(\mathbf{x}) = 418.9829n - \sum_{i=1}^{n} x_i \sin\left(\sqrt{|x_i|}\right)$$

Dixon-Price Function

$$f(\mathbf{x}) = (x_1 - 1)^2 + \sum_{i=2}^{n} i\left[2x_i^2 - x_{i-1}\right]^2$$

Michalewicz Function

$$f(\mathbf{x}) = -\sum_{i=1}^{n} \sin(x_i)\left[\sin\left(\frac{ix_i^2}{\pi}\right)\right]^{2m}$$

where m is set to 10.

Styblinski-Tang Function

$$f(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{n}\left[x_i^4 - 16x_i^2 + 5x_i\right]$$

2 Algorithms Implemented

This section discusses the implementation of the Particle Swarm Optimization (PSO) algorithm and explains how each component contributes to its effectiveness in solving optimization problems.

2.1 Particle Initialization

The `make_particles` function is responsible for initializing a swarm of particles, where each particle represents a candidate solution to the optimization problem. Key steps include:

- **Memory Allocation:** Each particle is allocated memory for its position, velocity, and personal best position. This allows dynamic handling of variables based on the problem's dimensionality (`NUM_VARIABLES`).

- **Random Initialization:** Particle positions are initialized randomly within specified bounds, ensuring diverse starting points for the search. Velocities are initially set to zero to enable controlled updates.
- **Evaluation of Initial Fitness:** The objective function evaluates each particle's position, and the corresponding fitness value is stored as the particle's personal best (`best_value`).

This initialization ensures a diverse population and sets the stage for iterative refinement during the optimization process.

2.2 Particle Swarm Optimization Algorithm

The `pso` function implements the core PSO algorithm. Its key components are:

2.2.1 Global and Personal Best Updates

- Each particle tracks its *personal best position* (`best_position`) and fitness value (`best_value`) based on the objective function's evaluations.
- The algorithm maintains a global best position (`best_position`) representing the best solution found by any particle.

2.2.2 Velocity and Position Updates

The velocity update equation incorporates:

$$v_{i,j} = w \cdot v_{i,j} + c_1 \cdot r_1 \cdot (p_{i,j} - x_{i,j}) + c_2 \cdot r_2 \cdot (g_j - x_{i,j})$$

where:

- w: Inertia weight to control the influence of previous velocity.
- c_1 and c_2: Acceleration coefficients governing attraction to personal and global best positions.
- r_1 and r_2: Random numbers in $[0, 1]$, introducing stochasticity.
- $p_{i,j}$: Personal best position of particle i in dimension j.
- g_j: Global best position in dimension j.

Each particle's position is updated based on its velocity, and values are clamped within specified bounds to ensure feasibility.

2.2.3 Convergence Check

The algorithm checks for convergence by monitoring improvements in the global best value. If the fitness value is below a predefined threshold (`THRESHOLD`), **which is the double precision: 1e-15**, the search terminates early.

2.3 Memory Management

To avoid memory leaks, all dynamically allocated resources (positions, velocities, and best positions) are freed after the algorithm completes.

2.4 Why It Works

The PSO algorithm mimics the social behavior of swarms, where particles iteratively refine their positions based on personal and collective experiences. By balancing exploration and exploitation, PSO is well-suited for optimizing complex, non-convex functions.

2.5 Optimizations for Rastrigin, Schwefel, and Rosenbrock Functions

To improve the performance of the Particle Swarm Optimization (PSO) algorithm for specific benchmark functions such as Rastrigin, Schwefel, and Rosenbrock, additional modifications were implemented:

- **Gradual Decrease of Inertia Weight (w):** The inertia weight w was gradually decreased over iterations. This strategy encourages global exploration during the initial stages and gradually shifts towards finer local exploitation, ensuring convergence to optimal solutions.
- **Dynamic Velocity Scaling:** If the velocity of a particle approached a certain threshold, its magnitude was doubled. This adjustment allowed the particles to explore higher-dimensional areas more effectively, which is particularly beneficial for multimodal functions like Rastrigin and Schwefel. This mechanism prevented premature convergence and enhanced the algorithm's ability to escape local optima.

These optimizations were tailored to address the unique challenges posed by the landscapes of these benchmark functions, ensuring a more robust and effective search process.

2.6 Output

The algorithm returns the global best fitness value and the corresponding position, providing an effective solution to the optimization problem. Additionally, convergence information is logged to help analyze the algorithm's performance.

3 How to Run the Code

The C code provided below should compile and run on any operating system that has a C compiler, such as GCC or Clang. First, open the relevant C files in your IDE or text editor, ensuring that the following files are located in the same directory: `main.c`, `utility.h`, `pso.c`, `OF.c`, `OF_lib.h`, and the `Makefile`.

To compile and run the code using GCC, follow these steps in the terminal:

1. Open the terminal and navigate to the directory containing the code files.
2. Type the following command to compile the code:

```
make
```

3. Once the compilation is successful, run the program with the following command:

```
./pos <Function Name> <Dimensions> <Lower Bound> <Upper Bound>
<Number of Particles> <Max Iterations>
```

Here, replace the placeholders with the appropriate values:

- `<Function Name>` is the name of the objective function you wish to use (e.g., Griewank, Levy, etc.).
- `<Dimensions>` is the number of variables (dimensions) in your problem.
- `<Lower Bound>` and `<Upper Bound>` are the bounds for the search space.
- `<Number of Particles>` is the number of particles to initialize in the Particle Swarm Optimization (PSO) algorithm.
- `<Max Iterations>` is the maximum number of iterations for the optimization process.

Once you run the program, it will execute the Particle Swarm Optimization algorithm and display the best solution found along with the corresponding fitness value.

Note: Ensure that all required dependencies, such as libraries and header files, are correctly linked in your project before running the program.

4 Case Study Tables

Table 1: `NUM_VARIABLES = 10` (or dimension $d = 10$) in **all** functions

Function	Bound		Particles	Iterations	Optimal Fitness	CPU time (Sec)
	Lower	Upper				
Griewank	-600	600	10000	784	0.000000	2.415697
Levy	-10	10	10000	157	0.000000	1.066276
Rastrigin	-5.12	5.12	10000	293	0.000000	1.808878
Rosenbrock	-5	10	10000	2500	0.000000	12.593930
Schwefel	-500	500	200000	188	0.000017	24.854368
Dixon-Price	-10	10	200000	168	0.000000	23.436315
Michalewicz	0	π	270000	336	-9.660152	110.630790
Styblinski-Tang	-5	5	10000	171	-391.661657	3.288109

Table 2: `NUM_VARIABLES = 50` (or dimension $d = 50$) in **all** functions

Function	Bound		Particles	Iterations	Optimal Fitness	CPU time (Sec)
	Lower	Upper				
Griewank	-600	600	10000	1000	0.000000	15.797975
Levy	-10	10	10000	1000	0.000000	15.241993
Rastrigin	-5.12	5.12	500000	9654	0.099495	1953.059442
Rosenbrock	-5	10	500000	7328	0.000000	9469.625316
Schwefel	-500	500	400000	4953	4.559432	2347.329323
Dixon-Price	-10	10	300000	789	0.000000	290.881407
Michalewicz	0	π	250000	10000	-20.883855	524.144172
Styblinski-Tang	-5	5	90000	10000	-1901.761409	62.078511

Table 3: `NUM_VARIABLES = 100` (or dimension $d = 100$) in **all** functions

Function	Bound		Particles	Iterations	Optimal Fitness	CPU time (Sec)
	Lower	Upper				
Griewank	-600	600	200000	1273	0.000000	374.956943
Levy	-10	10	200000	1648	0.000000	433.907699
Rastrigin	-5.12	5.12	700000	10000	40.66832	5897.729558
Rosenbrock	-5	10	700000	10000	23.233246	3019.304303
Schwefel	-500	500	700000	10000	267.654343	5434.322933
Dixon-Price	-10	10	200000	2341	0.666667	653.112404
Michalewicz	0	π	300000	10000	-83.518376	4332.830521
Styblinski-Tang	-5	5	600000	10000	-3492.514999	868.025572

5 Analysis of Optimization Functions and Performance

5.1 Global Minima Consistency

The optimization functions **Griewank**, **Dixon-Price**, and **Levy** exhibit consistent global minima across all dimensions. This property makes them reliable benchmarks for assessing optimization algorithms, as the global optima do not vary with the problem's dimensionality. Such consistency allows the focus to remain on the optimization process itself rather than changes in the function's landscape.

5.2 Dimensionality Effects on Other Functions

In contrast, the other five benchmark functions demonstrate a significant degradation in performance as the dimensionality increases. Higher dimensions expand the search space exponentially, making it more challenging for optimization algorithms to converge to the global minimum. The number of local minima grows rapidly with dimensionality, increasing the likelihood of particles being trapped in suboptimal regions and leading to poorer optimization results.

5.3 Time Complexity of Optimization

The computational time required to optimize these functions is directly proportional to the product of the number of particles and the number of iterations. Formally, the time complexity can be expressed as:

$$T \propto N_p \cdot N_i$$

where N_p is the number of particles and N_i is the number of iterations. Increasing either parameter improves the chances of finding the global minimum but also leads to a linear increase in computation time. Striking a balance between the number of particles, iterations, and dimensionality is essential to achieve efficient optimization.

6 Appendix

6.1 PSO.c

```
// CODE: include library(s)
#include "utility.h"
```

```
#include "OF_lib.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

// Helper function to generate random numbers in a range

double random_double(double min, double max) {
    return min + (max - min) * ((double)rand() / RAND_MAX);
}

// CODE: implement other functions here if necessary

void make_particles(Particle *particles, int NUM_PARTICLES, int
   NUM_VARIABLES, Bound *bounds, ObjectiveFunction objective_function) {

 for (int i = 0; i < NUM_PARTICLES; i++) {

  // Create particle
  Particle particle;

  // Allocate memory for position, velocity, and best_position
  particle.position = malloc(NUM_VARIABLES * sizeof(double));
  particle.velocity = malloc(NUM_VARIABLES * sizeof(double));
  particle.best_position = malloc(NUM_VARIABLES * sizeof(double));

  // Initialize position, velocity, and best_position
  for (int j = 0; j < NUM_VARIABLES; j++) {
   particle.position[j] = random_double(bounds[j].lowerBound, bounds[j].
      upperBound);
   particle.velocity[j] = 0.0;
  }

  // Get the value of the particle's position
  particle.value = objective_function(NUM_VARIABLES, particle.position);

  // Set the best value to the particle's current value
  particle.best_value = particle.value;

  // Add particle to the list of particles
  particles[i] = particle;
 }
```

```
}


double pso(ObjectiveFunction objective_function, const char *func_name, int
    NUM_VARIABLES, Bound *bounds, int NUM_PARTICLES, int MAX_ITERATIONS,
    double *best_position, double w, double c1, double c2) {
    // Allocate memory for particles
    Particle *particles = malloc(NUM_PARTICLES * sizeof(Particle));

    // Initialize particles
    make_particles(particles, NUM_PARTICLES, NUM_VARIABLES, bounds,
        objective_function);

    // Set initial best_position to the first particle's position
    for (int i = 0; i < NUM_VARIABLES; i++) {
        best_position[i] = particles[0].position[i];
    }

    // Variables for fitness tracking
    double previous_global_best = 1e15; // Large initial value
    int consecutive_count = 0;          // Counter for consecutive
        iterations
    int required_consecutive_iterations = 20; // Target consecutive count

    // Iterate for max iterations
    for (int i = 0; i < MAX_ITERATIONS; i++) {
        double global_best = objective_function(NUM_VARIABLES, best_position
            ); // Evaluate global fitness
        double w_max = w+0.2;
        double w_min = w-0.3;
        // Perform particle updates
        for (int p = 0; p < NUM_PARTICLES; p++) {
            // Update the velocity and position of the particle
            for (int j = 0; j < NUM_VARIABLES; j++) {

                // Define max_velocity as a fraction of the search space
                double max_velocity = (bounds[j].upperBound - bounds[j].
                    lowerBound) * 0.2;
                double w_new =w_max - (i/MAX_ITERATIONS)*(w_max-w_min);
                // Update position using velocity
                // Update velocity
                particles[p].velocity[j] = w * particles[p].velocity[j]
```

```
                + c1 * random_double(0, 1) * (particles[p].best_position
                    [j] - particles[p].position[j])
                + c2 * random_double(0, 1) * (best_position[j] -
                    particles[p].position[j]);

            // Clamp velocity Optimization attempt #1
            if((strcmp(func_name,"rastrigin") == 0)||(strcmp(func_name,"
                rosenbrock")==0)||(strcmp(func_name,"schwefel")==0)){
                if (particles[p].velocity[j] > max_velocity) {
                particles[p].velocity[j] = 2*max_velocity;
                } else if (particles[p].velocity[j] < -max_velocity) {
                particles[p].velocity[j] = -max_velocity*2;
                }
            }
            // Exploration vs Exploitation optimization
           if((strcmp(func_name,"rastrigin") == 0)||(strcmp(func_name,"
               rosenbrock")==0)||(strcmp(func_name,"schwefel")==0)){
                double w =w_max - (i/MAX_ITERATIONS)*(w_max-w_min);

            }

            // Update the position of the particle
            particles[p].position[j] += particles[p].velocity[j];

            // Clamp to within bounds
            if (particles[p].position[j] < bounds[j].lowerBound) {
                particles[p].position[j] = bounds[j].lowerBound;
            } else if (particles[p].position[j] > bounds[j].upperBound)
                {
                particles[p].position[j] = bounds[j].upperBound;
            }
        }

        // Update the value of the particle and the best value if
            necessary
        particles[p].value = objective_function(NUM_VARIABLES, particles
            [p].position);
        if (particles[p].value < particles[p].best_value) {
            particles[p].best_value = particles[p].value;
            for (int k = 0; k < NUM_VARIABLES; k++) {
                particles[p].best_position[k] = particles[p].position[k
                    ];
```

```
            }
        }

        // Update the global best value and best position if necessary
        if (particles[p].value < global_best) {
            global_best = particles[p].value;
            for (int k = 0; k < NUM_VARIABLES; k++) {
                best_position[k] = particles[p].position[k];
            }
        }
    }

    // Check fitness difference threshold for consecutive iterations
    if (fabs(global_best - previous_global_best) < THRESHOLD) {
        consecutive_count++;
        if (consecutive_count >= required_consecutive_iterations) {
            printf("Convergence due to fitness change threshold for %d
                consecutive iterations at iteration %d.\n",
                    required_consecutive_iterations, i + 1);
            break;
        }
    } else {
        consecutive_count = 0; // Reset counter if condition is not met
    }

    // Update previous global best fitness
    previous_global_best = global_best;

    // Check for overall convergence
     if (fabs(global_best) < THRESHOLD) {
        printf("Convergence reached after %d iterations.\n", i + 1);
        break;
    }


    // Debug output for each iteration
   //printf("Iteration %d: Best Fitness = %f\n", i + 1, global_best);
}

// Free memory
for (int i = 0; i < NUM_PARTICLES; i++) {
    free(particles[i].position);
```

```
        free(particles[i].velocity);
        free(particles[i].best_position);
    }
    free(particles);

    //printf("Using objective function: %s\n", func_name);
    return objective_function(NUM_VARIABLES, best_position); // Return the
        best fitness value
}
```

6.2 utility.h

```
#ifndef UTILITY_H
#define UTILITY_H


#define THRESHOLD 1e-15


//structs and typedefs
typedef double (*ObjectiveFunction)(int, double *);

typedef struct Bound{
    double lowerBound;
    double upperBound;
}Bound;

typedef struct Particle {

 double *velocity;
 double *position;
 double best_value;
 double *best_position;
 double value;

} Particle;

// Function prototypes
double random_double(double min, double max);
```

```
double pso(ObjectiveFunction objective_function,const char *func_name, int
    NUM_VARIABLES, Bound *bounds, int NUM_PARTICLES, int MAX_ITERATIONS,
    double best_position[], double w, double c1, double c2);
void make_particles(Particle *particles, int NUM_PARTICLES, int
    NUM_VARIABLES, Bound *bounds, ObjectiveFunction objective_function);
void update(Particle *particles, int NUM_PARTICLES, int NUM_VARIABLES,
    double *best_position, double w, double c1, double c2, ObjectiveFunction
    objective_function, Bound *bounds);

#endif // UTILITY_H
```

OPENAI 2024

University 2024a University 2024b. University 2024c University 2024d University 2024e University 2024f University 2024g University 2024h]

References

OPENAI (2024). *ChatGPT*. Accessed: 2024-11-30. URL: https://chat.openai.com/chat.

University, Simon Fraser (2024a). *Dixon Price Function*. Accessed: 2024-11-30. URL: https://www.sfu.ca/~ssurjano/dixonpr.html.

– (2024b). *Griewank Function*. Accessed: 2024-11-30. URL: https://www.sfu.ca/~ssurjano/griewank.html.

– (2024c). *Lévy Function*. Accessed: 2024-11-30. URL: https://www.sfu.ca/~ssurjano/levy.html.

– (2024d). *Michalewicz Function*. Accessed: 2024-11-30. URL: https://www.sfu.ca/~ssurjano/michal.html.

– (2024e). *Rastrigin Function*. Accessed: 2024-11-30. URL: https://www.sfu.ca/~ssurjano/rastr.html.

– (2024f). *Rosenbrock Function*. Accessed: 2024-11-30. URL: https://www.sfu.ca/~ssurjano/rosen.html.

– (2024g). *Schwefel Function*. Accessed: 2024-11-30. URL: https://www.sfu.ca/~ssurjano/schwef.html.

– (2024h). *Styblinski-Tang Function*. Accessed: 2024-11-30. URL: https://www.sfu.ca/~ssurjano/stybtang.html.