Assignment 4

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

Keerath Singh

Student Number 400506284

1 Introduction

Particle Swarm Optimization (PSO) is a computational algorithm used to solve complex optimization problems by simulating the collective behavior of particles in a swarm. Each particle represents a candidate solution and moves through the solution space by updating its position based on its own experience (personal best) and the best solution found by the swarm (global best).

In this assignment, PSO is applied to optimize several mathematical benchmark functions, such as Griewank, Rastrigin, Levy and etc... These functions present challenges like multiple local minima or high-dimensional search spaces. PSO's ability to balance exploration and exploitation makes it ideal for finding near-optimal solutions efficiently, even for problems where gradients or closed-form solutions are unavailable.

2 Benchmark Functions

2.1 Griewank Function

The Griewank function is a non-convex function with many local minima. It is used to test the global optimization capability of algorithms. The equation is:

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

2.2 Rastrigin Function

The Rastrigin function is highly multimodal and is often used to evaluate the exploration ability of optimization algorithms. The equation is:

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

2.3 Levy Function

The Levy function is designed to test the exploitation ability of optimization methods with its challenging landscape:

$$f(x) = \sin^2(\pi w_1) + \sum_{i=1}^{d-1}(w_i - 1)^2[1 + 10\sin^2(\pi w_i + 1)] + (w_d - 1)^2[1 + \sin^2(2\pi w_d)]$$

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

2.4 Rosenbrock Function

The Rosenbrock function, also known as the "banana function," is unimodal but with a narrow, curved valley. It is defined as:

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

2.5 Schwefel Function

The Schwefel function has numerous local minima and a deceptive global minimum near the edges of the search space. It is given by:

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

2.6 Dixon-Price Function

The Dixon-Price function is unimodal and designed to challenge optimization algorithms' exploitation capabilities:

$$f(x) = (x_1 - 1)^2 + \sum_{i=2}^{d} i(2x_i^2 - x_{i-1})^2$$

2.7 Michalewicz Function

The Michalewicz function has steep ridges and narrow valleys, testing the fine-tuning capability of optimization methods:

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

2.8 Styblinski-Tang Function

This function has many local minima, but its global minimum can be easily reached if the algorithm is well-tuned:

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

3 Particle Swarm Optimization (PSO) Formulation

$$v_{ij}(t+1) = w \cdot v_{ij}(t) + c_1 \cdot r_1 \cdot (p_{ij} - x_{ij}(t)) + c_2 \cdot r_2 \cdot (g_j - x_{ij}(t))$$

$$x_{ij}(t+1) = x_{ij}(t) + v_{ij}(t+1)$$

1. Evaluate the fitness of each particle at its new position: $f(\mathbf{x}_i(t+1))$.
2. Update personal best:

$$\mathbf{p}_i = \begin{cases} \mathbf{x}_i(t+1) & \text{if } f(\mathbf{x}_i(t+1)) < f(\mathbf{p}_i) \\ \mathbf{p}_i & \text{otherwise.} \end{cases}$$

3. Update global best:

$$\mathbf{g} = \arg\min_{\mathbf{p}_i} f(\mathbf{p}_i).$$

4 Running the Code

The PSO program can be compiled and run using the following steps:

4.1 Compilation

To compile the program, you can use the provided ‘Makefile‘ for convenience. Simply run the following command in your terminal:

```
make
```

This will generate an executable file named ‘pso‘ in your working directory.

If you prefer to compile manually, you can use the following command:

```
gcc -Ofast -o pso main.c PSO.c OF.c -lm
```

The ‘-Ofast‘ flag is used to enable aggressive compiler optimizations. This includes: - Enabling all the optimizations of the ‘-O3‘ level. - Ignoring strict standards compliance to allow faster math functions

These optimizations are particularly useful for computationally intensive tasks like PSO, where performance and speed are critical for handling large numbers of particles and iterations.

4.2 Running the Program

Once the compilation is successful, run the program with the following syntax:

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

For example, to run the PSO algorithm on the Griewank function with 8 dimensions, a lower bound of -50, an upper bound of 50, 500 particles, and 1000 iterations, use the command:

```
./pso griewank 8 -50 50 500 1000
```

4.3 Output

The program will output the optimal fitness value and the time taken to converge for the given function and parameters. Ensure the function names and bounds are correctly specified to avoid errors.

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	5000	5000	0.000000	1.744296
Levy	-10	10	200	1000	0.000000	0.019002
Rastrigin	-5.12	5.12	2000	1000	0.000000	0.197907
Rosenbrock	-5	10	50,000	1000	0.000000	8.273347
Schwefel	-500	500	50,000	10,000	0.000234	34.572465
Dixon-Price	-10	10	200	1000	0.666667	0.950392
Michalewicz	0	π	200000	10,000	-9.628880	3.305166
Styblinski-Tang	-5	5	20000	100	-391.473563	1.457899

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	50000	5000	0.000000	0.802035
Levy	-10	10	10000	1000	0.000000	1.928865
Rastrigin	-5.12	5.12	500000	10000	0.079895	2456.069645
Rosenbrock	-5	10	900000	10000	0.000000	1860.645326
Schwefel	-500	500	400000	10000	7.757672	4658.334323
Dixon-Price	-10	10	20000	1000	0.666667	11.831978
Michalewicz	0	π	250000	10000	-20.556335	1.173555
Styblinski-Tang	-5	5	90000	10000	-1647.050579	6.583283

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	2000	1000	0.000000	2.923729
Levy	-10	10	20000	10000	0.000000	68.054500
Rastrigin	-5.12	5.12	500000	10000	60.66892	3443.779855
Rosenbrock	-5	10	500000	10000	36.799256	2624.304303
Schwefel	-500	500	700000	10000	235.715348	6943.322933
Dixon-Price	-10	10	20000	2000	0.666667	153.112404
Michalewicz	0	π	300000	10000	-32.228714	6.754527
Styblinski-Tang	-5	5	200000	10000	-3294.885820	15.536366

5 References

1. OpenAI, "ChatGPT," OpenAI, San Francisco, CA, 2024. [Online]. Available: https://openai.com/chatgpt. [Accessed: Dec. 5, 2024].

2. S. Surjanovic and D. Bingham, "Dixon-Price Function," Simon Fraser University, Burnaby, BC. [Online]. Available: https://www.sfu.ca/~ssurjano/dixonpr.html. [Accessed: Dec. 5, 2024].

6 Appendix

PSO.c

```
  #include <stdlib.h>
#include <math.h>
#include <float.h>
#include <stdio.h> // For printf
#include "utility.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);
}

// Function to initialize particles --> time complexity is O(n^2), causing
   more time to run
void create_p(Particle *particles, int NUM_PARTICLES, int NUM_VARIABLES,
   Bound *bounds, ObjectiveFunction objective_function, double *
   global_best_position, double *global_best_value) {
    for (int i = 0; i < NUM_PARTICLES; i++) {
        // Allocate memory for particle attributes, thus creating them
        particles[i].position = malloc(NUM_VARIABLES * sizeof(double));
        particles[i].velocity = malloc(NUM_VARIABLES * sizeof(double));
        particles[i].best_position = malloc(NUM_VARIABLES * sizeof(double));
        particles[i].best_value = DBL_MAX;

        // Assign teh particle position and velocity
        for (int j = 0; j < NUM_VARIABLES; j++) {
            particles[i].position[j] = random_double(bounds[j].lowerBound,
               bounds[j].upperBound);
            particles[i].velocity[j] = random_double(-1.0, 1.0);
            particles[i].best_position[j] = particles[i].position[j];
        }

        particles[i].value = objective_function(NUM_VARIABLES, particles[i].
           position);
        if (particles[i].value < *global_best_value) {
            *global_best_value = particles[i].value;
            for (int j = 0; j < NUM_VARIABLES; j++) {
                global_best_position[j] = particles[i].position[j];
            }
```

```
        }
    }
}

// Function to update particle velocities and positions
void repos_p(Particle *particles, int NUM_PARTICLES, int NUM_VARIABLES,
   Bound *bounds,
                    ObjectiveFunction objective_function, double *
                       global_best_position, double *global_best_value,
                       double w, double c1, double c2) {
    for (int i = 0; i < NUM_PARTICLES; i++) {
        for (int j = 0; j < NUM_VARIABLES; j++) {
            double r1 = random_double(0.0, 1.0);
            double r2 = random_double(0.0, 1.0);

            // Update velocity
            particles[i].velocity[j] = w * particles[i].velocity[j]
                                       + c1 * r1 * (particles[i].best_position
                                          [j] - particles[i].position[j])
                                       + c2 * r2 * (global_best_position[j] -
                                          particles[i].position[j]);

            // Clamp velocity --> CHATGPT was used to help come up with this
                idea of clamping velocities
            double max_velocity = (bounds[j].upperBound - bounds[j].
               lowerBound) * 0.2;
            if (particles[i].velocity[j] > max_velocity) particles[i].
               velocity[j] = max_velocity;
            if (particles[i].velocity[j] < -max_velocity) particles[i].
               velocity[j] = -max_velocity;

            // Update position
            particles[i].position[j] += particles[i].velocity[j];

            // Reflect position if out of bounds
            if (particles[i].position[j] < bounds[j].lowerBound) {
                particles[i].position[j] = bounds[j].lowerBound + (bounds[j
                   ].lowerBound - particles[i].position[j]);
                particles[i].velocity[j] *= -1;  // Reverse direction
            }
            if (particles[i].position[j] > bounds[j].upperBound) {
                particles[i].position[j] = bounds[j].upperBound - (particles
```

```
                    [i].position[j] - bounds[j].upperBound);
                particles[i].velocity[j] *= -1;  // Reverse direction
            }
        }

        // Evaluate fitness
        particles[i].value = objective_function(NUM_VARIABLES, particles[i].
            position);

        // Update personal best
        if (particles[i].value < particles[i].best_value) {
            particles[i].best_value = particles[i].value;
            for (int j = 0; j < NUM_VARIABLES; j++) {
                particles[i].best_position[j] = particles[i].position[j];
            }
        }

        // Update global best
        if (particles[i].value < *global_best_value) {
            *global_best_value = particles[i].value;
            for (int j = 0; j < NUM_VARIABLES; j++) {
                global_best_position[j] = particles[i].position[j];
            }
        }
    }
}

// Free memory allocated for particles
void free_particles(Particle *particles, int NUM_PARTICLES) {
    for (int i = 0; i < NUM_PARTICLES; i++) {
        free(particles[i].position);
        free(particles[i].velocity);
        free(particles[i].best_position);
    }
}

// PSO implementation
double pso(ObjectiveFunction objective_function, int NUM_VARIABLES, Bound *
    bounds, int NUM_PARTICLES, int MAX_ITERATIONS, double *best_position) {
    // Allocate memory for particles and global best variables
    Particle *particles = malloc(NUM_PARTICLES * sizeof(Particle));
    double global_best_value = DBL_MAX;
```

```
    double *global_best_position = malloc(NUM_VARIABLES * sizeof(double));

    // Initialize particles
    create_p(particles, NUM_PARTICLES, NUM_VARIABLES, bounds,
        objective_function, global_best_position, &global_best_value);

    // Main PSO loop
    double w = 0.7, c1 = 1.5, c2 = 1.5; // PSO hyperparameters
    double TS = 1e-13;                   // threshold --> Pedram talked
        about this in class
    int patience = 50;                   // Iterations to wait for
        improvement
    int convergence_counter = 0;         // Counter for consecutive non-
        improvement iterations

    for (int iter = 0; iter < MAX_ITERATIONS; iter++) {
        repos_p(particles, NUM_PARTICLES, NUM_VARIABLES, bounds,
            objective_function,
                        global_best_position, &global_best_value, w, c1, c2
                            );

        // Early stopping condition
        if (global_best_value < TS) {
            convergence_counter++;
            if (convergence_counter >= patience) {
                printf("Stopping early after %d iterations, fitness: %lf\n",
                    iter + 1, global_best_value);
                break;
            }
        } else {
            // Reset counter if improvement is observed
            convergence_counter = 0;
        }
    }

    // Copy global best position to output
    for (int i = 0; i < NUM_VARIABLES; i++) {
        best_position[i] = global_best_position[i];
    }

    // Free memory
    free_particles(particles, NUM_PARTICLES);
```

```
    free(particles);
    free(global_best_position);

    return global_best_value;
}
```

Utility.h

```
#ifndef UTILITY_H
#define UTILITY_H

// Function pointer type for objective functions
typedef double (*ObjectiveFunction)(int, double *);

// Structure for bounds of each variable
typedef struct Bound {
    double lowerBound; // The lowr limit for a variable in the search space
    double upperBound; // The upper limit for a variable in the search space
} Bound;

// Structure for a single particle
typedef struct Particle {
    double *position;       // Current postion of the particle in the serch
       space
    double *velocity;       // Current velocioty or direction of movement
    double *best_position;  // The personal best postion of this particle
    double value;           // Fitness value of the current postion
    double best_value;      // Best fitness value ever achieved by the
       particle
} Particle;

// Function prototypes

// Helper function to generate a random double between min and max.
// NOTE: Random values are used to create diversitiy in particle movement
double random_double(double min, double max);

// Function to initialize all particles
// Sets their positions within the search space bounds and assigns small
   random velocities
// The glboal best position and value are also updated if the initialized
   particle has better fitness

//CHATGPT was used to help come up with function parameters
void create_p(Particle *particles, int NUM_PARTICLES, int NUM_VARIABLES,
   Bound *bounds, ObjectiveFunction objective_function, double *
   global_best_position, double *global_best_value);
```

```
// Updates the velocities and positions of particles
// Adjusts based on personal and global bests using the PSO formula
// Also clamps the velocities to prevent "too fast" movements, which could
   break convergence
// CHATGPT was used to help come up with this idea of clamping velocities

//CHATGPT was used to help come up with function parameters
void repos_p(Particle *particles, int NUM_PARTICLES, int NUM_VARIABLES,
   Bound *bounds,
                    ObjectiveFunction objective_function, double *
                       global_best_position, double *global_best_value,
                       double w, double c1, double c2);

// Free the dynamically allocated memory for particle attributes
void free_particles(Particle *particles, int NUM_PARTICLES);

// The main PSO function
// Orchestrates the whole process:
// - Initializes particles
// - Iteratively updates particles' velocities and positions
// - Tracks the global best solution
// Returns the fitness of the best solution found after all iterations
double pso(ObjectiveFunction objective_function, int NUM_VARIABLES, Bound *
   bounds, int NUM_PARTICLES, int MAX_ITERATIONS, double *best_position);


#endif
```