Assignment 4

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

Fady Abousifein

400506836

1 Introduction

In this assignment we are tasked with implementing **Particle Swarm Optimization (PSO)**, which is a method of optimization modeled after the social behavior of birds flocking or fish schooling. It is often used when solving optimization problems at which normal mathematical methods may eithe not work or be too computationally taxing.

PSO simulates the behavior of a large amount of particles each of which represent a possible solution to the optimization problem. These particles iterativley change their position based on their best positions and the global best position of all of the particles, eventually the swarm of particles will converge towards one position and thus the optimal solution.

2 Particle Swarm Optimization (PSO) Formulation

Before we start talking about how the algorithim works, we must first define the constants and variables that we will be using:

- N: Number of particles in the swarm.
- D: Dimensionality of the search space (number of variables).
- $\mathbf{x}_i = \begin{bmatrix} x_{i1}, x_{i2}, \ldots, x_{iD} \end{bmatrix}^\top$: Position vector of the ith particle.
- $\mathbf{v}_i = \begin{bmatrix} v_{i1}, v_{i2}, \ldots, v_{iD} \end{bmatrix}^\top$: Velocity vector of the ith particle.
- p_i: Personal best position of the ith particle.
- g: Global best position among all particles.
- w: Inertia weight, controlling the influence of the previous velocity.
- c_1, c_2: Cognitive coefficient, measuring the particle's tendency to return to its personal best position, and the Social coefficient, measuring the particle's tendency to follow the global best position.

- r_1, r_2: Random numbers uniformly distributed in $[0, 1]$.

These are all the variables and constants that are significant to the PSO algorithm. The PSO algorithm is dependent on how the particles update their attributes, and these the the formulas that define how the particles update their attributes:

Velocity Update

The velocity of each particle is updated at each iteration as follows:

$$\mathbf{v}_i^{(t+1)} = w\mathbf{v}_i^{(t)} + c_1 r_1 \left(\mathbf{p}_i - \mathbf{x}_i^{(t)}\right) + c_2 r_2 \left(\mathbf{g} - \mathbf{x}_i^{(t)}\right)$$

Position Update

The position of each particle is updated as follows:

$$\mathbf{x}_i^{(t+1)} = \mathbf{x}_i^{(t)} + \mathbf{v}_i^{(t+1)}$$

Objective Function

After each iteration the particle's fitness value is calculated using the objective function.

Best Positions

- The **personal best position**, $\mathbf{p}_i$, is updated whenever a particle achieves a better value for the objective function:

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

- The **global best position**, $\mathbf{g}$, is updated to be the best position among all particles:

$$\mathbf{g} = \arg \min_{i \in \{1,\ldots,N\}} f(\mathbf{p}_i).$$

Stopping Criteria

The above steps summarize the entire algorithm, however, when do we actually stop the iterations? In the assignment we are told that these are the following termination conditions:

- The maximum number of iterations, $T_{\max}$, is reached.
- The fitness value of the global best position falls below a predefined threshold, ϵ which is double precision.

Notice that unlike the assignment I am using vector notation, thus eliminating the j components of the particle attributes. This was done simply to make the equations more compact, however, the code itself will use the scalar components of the vectors.

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	0.0196	0.0387
Levy	-10	10	200	1000	2.8179	0.013285
Rastrigin	-5.12	5.12	2000	1000	21.8890	0.011233
Rosenbrock	-5	10	50,000	1000	0.2864	2.1153
Schwefel	-500	500	50,000	10,000	1191.9687	0.1291
Dixon-Price	-10	10	200	1000	0.6667	0.0080
Michalewicz	0	π	200000	10,000	-4.3275	0.3551
Styblinski-Tang	-5	5	20000	100	-320.9715	0.0205

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	0.9745
Levy	-10	10	10000	1000	13.7100	0.3406
Rastrigin	-5.12	5.12	500000	10000	153.2942	3.5899
Rosenbrock	-5	10	900000	10000	25052.9603	2.8546
Schwefel	-500	500	400000	10000	8944.0695	3.1870
Dixon-Price	-10	10	20000	1000	321.0000	0.2272
Michalewicz	0	π	250000	10000	-21.6375	3.4989
Styblinski-Tang	-5	5	90000	10000	-1717.9840	2.2758

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	0.0034	0.6769
Levy	-10	10	20000	10000	43.0579	6.4120
Rastrigin	-5.12	5.12	500000	10000	331.5333	6.9625
Rosenbrock	-5	10	500000	10000	99.4779	4.4570
Schwefel	-500	500	700000	10000	16241.0766	7.2933
Dixon-Price	-10	10	20000	2000	85.8870	1.3726
Michalewicz	0	π	300000	10000	-48.6944	8.3429
Styblinski-Tang	-5	5	200000	10000	-3450.1048	6.0242

3 Running the Program

Simply run the Makefile by running "make" then you can run the executable pso by running:

```
./pso <Function name> <Dimensions> <Lower Bound> <Upper Bound>
  <Number of Particles> <Max iterations>
```

4 Appendix

4.1 PSO.c

```
 // CODE: include library(s)
#include "utility.h"
#include "OF_lib.h"
#include <stdio.h>
#include <stdlib.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

// Function which will initialize a particle and all its attributes as seen
   in the psuedo code
void initializer(Particle * particles, int NUM_PARTICLES, int NUM_VARIABLES,
    Bound * bounds, ObjectiveFunction objectiveFunction) {
    for (int i = 0; i < NUM_PARTICLES; i++) {
        // declare a new particle
        Particle particle;

        // allocate  memory for a particles attributes defined in the
           Particle struct
        particle.position = malloc(NUM_VARIABLES * sizeof(double));
        particle.velocity = malloc(NUM_VARIABLES * sizeof(double));
        particle.bestPosition = malloc(NUM_VARIABLES * sizeof(double));
```

```
        // initialize values for the particles above attributes
        for (int j = 0; j < NUM_VARIABLES; j++) {
            particle.position[j] = random_double(bounds[j].lowerBound,
                bounds[j].upperBound);
            particle.velocity[j] = 0.0;
            // the bestPosition will be initialized in PSO
        }

        // retrieve particles current position and initialize the best value
            as the current value
        particle.value = objectiveFunction(NUM_VARIABLES, particle.position)
            ;
        particle.bestValue = particle.value;

        particles[i] = particle; // this particle has now been added to the
            list of particles
    }
}

// Function which will update a particles attributes throughout the
    algorithm
void attributeUpdates(Particle * particles, int NUM_PARTICLES, int
    NUM_VARIABLES, double * bestPosition, double w, double c1, double c2,
    ObjectiveFunction objectiveFunction, Bound * bounds) {
    // declare and initialize the global best value
    double globalBest = objectiveFunction(NUM_VARIABLES, bestPosition);

    // update particle attributes
    for (int i = 0; i < NUM_VARIABLES; i++) {
        for (int j = 0; j < NUM_VARIABLES; j++) {
            // velocity update
            particles[i].velocity[j] = w * particles[i].velocity[j] + c1 *
                random_double(0, 1) * (particles[i].bestPosition[j] -
                particles[i].position[j]) + c2 * random_double(0, 1) * (
                bestPosition[j] - particles[i].position[j]);

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

            // clamp the particles position within the bounds as defined in
                the PseudoCode
            if (particles[i].position[j] < bounds[j].lowerBound) {
```

```
                particles[i].position[j] = bounds[j].lowerBound;
            }
            else if (particles[i].position[j] > bounds[j].upperBound) {
                particles[i].position[j] = bounds[j].upperBound;
            }
        }

        // particle value update
        particles[i].value = objectiveFunction(NUM_VARIABLES, particles[i].
            position);

        // best value/position update if needed
        if (particles[i].value < particles[i].bestValue) {
            particles[i].bestValue = particles[i].value;
            for (int k = 0; k < NUM_VARIABLES; k++) {
                particles[i].bestPosition[k] = particles[i].position[k];
            }
        }

        // global best value/position update if needed
        if (particles[i].value < globalBest) {
            globalBest = particles[i].value;
            for (int k = 0; k < NUM_VARIABLES; k++) {
                bestPosition[k] = particles[i].position[k];
            }
        }

    }

}

// pso implementation
double pso(ObjectiveFunction objectiveFunction, int NUM_VARIABLES, Bound *
    bounds, int NUM_PARTICLES, int MAX_ITERATIONS, double *bestPosition) {
    // declare and initialize threshold to double precision
    double threshold = 1.0e-15;

    // allocate memmory for all particles and initialize them with their
        attributes
    Particle * particles = malloc(NUM_PARTICLES * sizeof(Particle));
    initializer(particles, NUM_PARTICLES, NUM_VARIABLES, bounds,
        objectiveFunction);
```

```
    // initialize the bestPosition to the first particles position (this is
       just a placeholder essentially)
    for (int i = 0; i < NUM_VARIABLES ; i++) {
       bestPosition[i] = particles[0].position[i];
    }

    // update attributes of the particles
    for (int i = 0; i < MAX_ITERATIONS; i++) {
        attributeUpdates(particles, NUM_PARTICLES, NUM_VARIABLES,
           bestPosition, 0.7, 1.5, 1.5, objectiveFunction, bounds);
    }

    // free the allocated memory
    for (int i = 0; i < NUM_PARTICLES; i++) {
        free(particles[i].position);
        free(particles[i].velocity);
        free(particles[i].bestPosition);
    }
    free(particles);

    return objectiveFunction(NUM_VARIABLES, bestPosition);
}
```

4.2 utility.h

```
#ifndef UTILITY_H
#define UTILITY_H

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

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


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

```
double pso(ObjectiveFunction objective_function, int NUM_VARIABLES,
    Bound *bounds, int NUM_PARTICLES, int MAX_ITERATIONS, double
   best_position[]);

// CODE: declare other functions and structures if necessary
typedef struct Particle {
    double * position;
    double * velocity;
    double * bestPosition;
    double bestValue;
    double value;
} Particle;

void initializer(Particle * particles, int NUM_PARTICLES, int
   NUM_VARIABLES, Bound * bounds, ObjectiveFunction
   objectiveFunction);
void attributeUpdates(Particle * particles, int NUM_PARTICLES, int
   NUM_VARIABLES, double * bestPosition, double w, double c1,
   double c2, ObjectiveFunction objectiveFunction, Bound * bounds);

#endif // UTILITY_H
```