Assignment 4

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

Elijah Cosby

400538072

1 Introduction

This assignment was to create an optimization algorithm using a Particle Swarm Optimization (PSO) method. This is a method of finding the best values for an amount of variables (dimensions) to satisfy a function that represents the necessary components and constraints of a problem that needs to be solved, otherwise known as an optimization problem. The PSO method works by mimicking how a school of fish solve a problem, as in everyone helps each other out a little bit by sharing their good ideas on how to solve a problem and allowing everyone else to build off of each other's good ideas to more quickly come to a communal consensus of what the best idea to solve the problem is. It's a testamate to the power of collaboration and how 2 heads (or 20000 heads haha) are better than 1. I found this assignment, and the concept of optimization algorithms, quite fascinating! I wish this wasn't around exam season so I would have been able to spend even more time experimenting with various methodologies of how to get some of the most complex functions for 50 or 100 dimensions.

2 How to Run

1. Compiling and linking:

   ```
   make
   ```

2. Running the code - format:

   ```
   ./pso <ObjectiveFunctionName> <NUM_VARIABLES> <LowerBound> <UpperBound> <NUM_PARTICLES> <MAX_ITERATIONS>
   ```

3 Explanation of Functions + Methods Used

allocateInitParticles

This function allocates memory for all the particles, assigns initial random values to each particles position, velocity, and best position attributes, calculates each particle's fitness value, and then figures out which particle has the best fitness value out of the entire swarm of particles. The Particle structure I created has 5 members which consist of its position, velocity, its best position, its fitness values, and its best fitness value. I'll get into how these are used in the PSO in the explanation of the PSO itself.

PSO

For the most part my PSO's structure was similar to the pseudo code provided to us in the instructions; however, I made some changes to try an make the PSO a little more efficient. For instance, I used a local best ring topology method rather than the global best method to try and better avoid some local minima, and I also instated a stop condition so that the iteration loop would break after 100 iterations with a constant best fitness value. The PSO function works by having a bigger iterations loop that basically controls how many times each particle has moved, and it doesn't let the same particle move again until every other particle has moved to its new place. Within this iteration loop, there's then a loop which basically figures out how well each particle is doing, and if it's doing better than it previously has, it holds onto it's personal best value and position it was at when it had this best value. Next is another loop that this time finds which particle had the best value between each of a particle's neighbours in the array, it then stores this variable as that particle's local best within an array to be accessed later by that particle. Next loop is where all of the particles are launched to their new positions, and this is similarly to how it was done in the pseudo code; however, since I was doing it the local best way, I used the local best varible we found for the particle being moved that loop through from the previous loop. I also added clamping conditions to make sure that the particles stay within the specified bounds. It then updates the current best if needed, and then checks the stopping conditions (which are basically checking to see if the best global fitness value has changed within the past 100 iterations), and then finally once out of the iteration loop it assigns the values of the particle that had the best position to that corresponding variable that was given as a parameter to the function via a pointer, and finally retruns the optimal fitness value.

*Notes:

1. The way I set up my stopping criteria ensured a much more precise result than simply the 4 decimal points we were told to provide, so the CPU time and the number of iterations may be on the longer/larger side.

2. Blank means I was unable to get the optimal value for this function.

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	20000	778	0.0000	6.86
Levy	-10	10	20000	1056	0.0000	8.24
Rastrigin	-5.12	5.12	50000	1023	0.0000	24.97
Rosenbrock	-5	10	20000	1999	0.0000	13.19
Schwefel	-500	500	50000	966	0.0001	26.22
Dixon-Price	-10	10	50000	1879	0.0000	28.74
Michalewicz	0	π	50000	1934	-9.66015	71.34
Styblinski-Tang	-5	5	50000	621	-391.66	14.09

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	2817	0.0000	327.85
Levy	-10	10	20000	5367	0.0000	237.95
Rastrigin	-5.12	5.12				
Rosenbrock	-5	10	100000	23906	0.0000	3926.72
Schwefel	-500	500				
Dixon-Price	-10	10	10000	17937	0.0000	486.4
Michalewicz	0	π				
Styblinski-Tang	-5	5				

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	100000	5417	0.0000	2245.14
Levy	-10	10				
Rastrigin	-5.12	5.12				
Rosenbrock	-5	10	2500	149833	0.0000	1345.18
Schwefel	-500	500				
Dixon-Price	-10	10				
Michalewicz	0	π				
Styblinski-Tang	-5	5				

4 Appendix

```
  //PSO.c
        // CODE: include library(s)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "OF_lib.h"
#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);
}


// CODE: implement other functions here if necessary
void allocateInitParticles(Particle *Swarm, int NUM_PARTICLES, int
   NUM_VARIABLES, Bound *bounds, ObjectiveFunction objective_function,
   double *bestGlobalFitness, double *bestGlobalPosition) {
    for(int i=0; i<NUM_PARTICLES;i++){
        Swarm[i].position = malloc(NUM_VARIABLES*sizeof(double));
        Swarm[i].velocity = malloc(NUM_VARIABLES*sizeof(double));
        Swarm[i].bestPosition = malloc(NUM_VARIABLES*sizeof(double));
        for(int j=0; j<NUM_VARIABLES;j++){
            Swarm[i].position[j] = random_double(bounds[j].lowerBound,
                bounds[j].upperBound);
            Swarm[i].velocity[j] = random_double(bounds[j].lowerBound,
                bounds[j].upperBound);
            Swarm[i].bestPosition[j] = Swarm[i].position[j];
        }
        Swarm[i].fitnessVal = objective_function(NUM_VARIABLES, Swarm[i].
            position);
        Swarm[i].bestFitnessVal = Swarm[i].fitnessVal;
        if (Swarm[i].bestFitnessVal < *bestGlobalFitness){
            *bestGlobalFitness = Swarm[i].fitnessVal;
            for(int j=0; j<NUM_VARIABLES;j++){
                bestGlobalPosition[j] = Swarm[i].bestPosition[j];
            }
        }
    }
}


double pso(ObjectiveFunction objective_function, int NUM_VARIABLES, Bound *
   bounds, int NUM_PARTICLES, int MAX_ITERATIONS, double *best_position) {
//Initializes constants and then particles
    const double COG_COEF = 1.5;
    const double GLOB_COEF = 1.5;
    const double WEIGHT = 0.7;
    const int MAX_NO_IMPROV = 100;
    double bestGlobalFitness = INFINITY;
    double previousBest = bestGlobalFitness;
    int noImprovement = 0;
    double *bestGlobalPosition = malloc(NUM_VARIABLES * sizeof(double));
    Particle *Swarm = (Particle *)malloc(NUM_PARTICLES*sizeof(Particle));

    allocateInitParticles(Swarm, NUM_PARTICLES, NUM_VARIABLES, bounds,
       objective_function, &bestGlobalFitness, bestGlobalPosition);

//For neighboorhood method, adding a local best
    double *bestLocalPosition = malloc(NUM_PARTICLES * NUM_VARIABLES *
       sizeof(double));
    for (int i = 0; i < NUM_PARTICLES; i++) {
        for (int k = 0; k < NUM_VARIABLES; k++) {
            bestLocalPosition[i * NUM_VARIABLES + k] = Swarm[i].bestPosition
               [k];
        }
    }

//Now onto the actual PSO
    for (int i=0; i<MAX_ITERATIONS; i++){


        for(int j=0; j<NUM_PARTICLES;j++){
            Swarm[j].fitnessVal = objective_function(NUM_VARIABLES,Swarm[j].
               position);
            if (Swarm[j].fitnessVal < Swarm[j].bestFitnessVal){
                Swarm[j].bestFitnessVal = Swarm[j].fitnessVal;
                for(int k=0; k<NUM_VARIABLES;k++){
                    Swarm[j].bestPosition[k] = Swarm[j].position[k];
                }
            }
        }

//Gets local best for each particle
        for(int j=0; j<NUM_PARTICLES; j++){
            int right;
            if (j == NUM_PARTICLES - 1){
                right = 0;
            } else {
                right = j + 1;
            }
            int left;
            if (j == 0){
                left = NUM_PARTICLES - 1;
            } else {
                left = j - 1;
            }
            double bestFitness = Swarm[j].bestFitnessVal;
            int bestIndex = j;
            if (Swarm[left].bestFitnessVal < bestFitness) {
                bestFitness = Swarm[left].bestFitnessVal;
                bestIndex = left;
            }
            if (Swarm[right].bestFitnessVal < bestFitness) {
                bestFitness = Swarm[right].bestFitnessVal;
                bestIndex = right;
            }

            for (int k = 0; k < NUM_VARIABLES; k++) {
                bestLocalPosition[j * NUM_VARIABLES + k] = Swarm[bestIndex].
                    bestPosition[k];
            }
        }

        for (int j=0; j<NUM_PARTICLES; j++){
            for (int k=0; k<NUM_VARIABLES; k++){
                double r1 = random_double(0, 1);
                double r2 = random_double(0, 1);
                Swarm[j].velocity[k] = WEIGHT*Swarm[j].velocity[k]+COG_COEF*
                    r1*(Swarm[j].bestPosition[k]-Swarm[j].position[k])+
                    GLOB_COEF*r2*(bestLocalPosition[k+(j*NUM_VARIABLES)]-
                    Swarm[j].position[k]);
                Swarm[j].position[k] = Swarm[j].position[k]+Swarm[j].
                    velocity[k];
                if (Swarm[j].position[k] < bounds[k].lowerBound){
                    Swarm[j].position[k] = bounds[k].lowerBound;
                }
                if (Swarm[j].position[k] > bounds[k].upperBound){
                    Swarm[j].position[k] = bounds[k].upperBound;
                }
            }

            if (Swarm[j].fitnessVal < bestGlobalFitness){
                bestGlobalFitness = Swarm[j].fitnessVal;
                for(int k=0; k<NUM_VARIABLES;k++){
                    bestGlobalPosition[k] = Swarm[j].position[k];
                }
            }
        }
    double currentBest = Swarm[0].bestFitnessVal;
    for(int j = 1; j < NUM_PARTICLES; j++){
        if (Swarm[j].bestFitnessVal < currentBest){
            currentBest = Swarm[j].bestFitnessVal;
        }
    }
    bestGlobalFitness = currentBest;
    printf("Iteration %d, Best Fitness: %f\n", i, bestGlobalFitness);
    if(bestGlobalFitness < previousBest){
        noImprovement=0;
    } else{
        if (noImprovement > MAX_NO_IMPROV){
            printf("No improvement, stopping @ %d iterations.\n", i);
            break;
        }
        noImprovement++;
    }
    previousBest = bestGlobalFitness;
    }

    for (int i=0; i<NUM_VARIABLES; i++){
        best_position[i] = bestGlobalPosition[i];
    }
```

```
//Freeing the memory used
    for (int i=0; i<NUM_PARTICLES; i++){
        free(Swarm[i].position);
        free(Swarm[i].velocity);
        free(Swarm[i].bestPosition);
    }
    free(Swarm);
    free(bestGlobalPosition);
    free(bestLocalPosition);
    //free(bestLocalFitness);
    return bestGlobalFitness;
}

//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 fitnessVal;
    double bestFitnessVal;
}Particle;

#endif // UTILITY_H
```

Citations: A. P. Engelbrecht, "Particle Swarm Optimization: Global Best or Local Best?," 2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence, Ipojuca, Brazil, 2013, pp. 124-135, doi: 10.1109/BRICS-CCI-CBIC.2013.31. keywords: Accuracy;Topology;Convergence;Benchmark testing;Algorithm design and analysis;Optimization;Computatio intelligence;Particle swarm optimization;neigbhorhood topologies;global best;local best,

-As well as chatGPT for some edits and Youtube for understanding concepts