2MP3 Assignment 4

Garret Langlois

December 2024

1 Introduction

One of the largest problems to solve in industry with regards to Engineering is the task of optimization. Traditionally, this is done with methods like Gradient Descent. However, with regards to functions with many local minima it may be difficult to find the minimum possible value. This is where the benefit of **Particle Swarm Optimization (PSO)** is. This allows us to place many [**particles**] over the domain of the function we wish to minimize and allow them to trend towards a global minima.

2 Problem Statement

In this assignment I developed a PSO algorithm with the task of minimizing the following functions:

1. Griewank
2. Levy
3. Rastrigin
4. Rosenbrock
5. Schwefel
6. Dixon-Price
7. Michalewicz with m=10
8. Styblinksi-Tang

This optimization had to be carried out over an arbitrary number of dimensions, denoted in an input array of **x**. The values of x are modified through a velocity function which is calculated at each iteration of the function. The max number of iterations that is allowed is defined by the user.

3 Solution

My solution mostly followed the layout from the assignment pdf, however I did make a few notable changes to ensure convergence with higher dimension versions of the more complex functions.
All of the logic for the algorithm is controlled by the **PSO()** function inside the **PSO.c** file. Below I have separated the development of the algorithms into sections.

3.1 Random Number Generation

One of the key changes that I made when developing this algorithm when compared to the original layout was to swap out the default random number generation logic. The original random number generating function in the file did not work properly on my computer so I instead decided to implement the **drand48()** random number generation function. The implementation for this is the following:

```c

// New random function because the old one wouldn't work for me, this uses drand48
double random_double(double min, double max) {

    //Create a random fraction from 0 to 1
    double random_fraction = drand48();

    //Return the random number that is generated within the bounds
    return min+ (max-min)*random_fraction;

}
```

3.2 Particle Initialization

When first creating the particles it is important that they are initialized, this is done by looping through all of the particles and assigning a random position within the bounds of the function for them to be placed at. From there the algorithm can iteratively move the particles towards the global minima. Below is the implementation of the initialization part of the **PSO.c** function:

```c
    //Initialize all of the dimensions for each particle
    for(int i = 0; i < NUM_PARTICLES; i++) {


        //Malloc everything
        particleArray[i]->decisionVariables = (double *)malloc(NUM_VARIABLES*sizeof(double));
        particleArray[i]->personalBest = (double *)malloc(NUM_VARIABLES*sizeof(double));
        particleArray[i]->velocityVariables = (double *)malloc(NUM_VARIABLES*sizeof(double));

        //Iterate through all of the dimensions and set the values to random numbers
        for(int j=0; j < NUM_VARIABLES; j++) {
            double random_num = random_double(bounds->lowerBound, bounds->upperBound);

            //We want to set the decision variables and the personal best to the same value, that
                way we can iterate on them
            particleArray[i]->decisionVariables[j] = random_num;
            particleArray[i]->personalBest[j] = random_num;

            //We set the velocity to a random number between -1 and 1
            particleArray[i]->velocityVariables[j] = random_double(-1, 1);
        }


        //Calculate the current position of the particle and set it as both the current position
            and the personal best fitness
        //We do this because the first iteration will always be the best
        double currentPosition = objective_function(NUM_VARIABLES, particleArray[i]->
            decisionVariables);
        particleArray[i]->currentPosition = currentPosition;
        particleArray[i]->personalBestFitness = currentPosition;

        //Keep track of the global best position from the initialized particles
        if (currentPosition < globalBestPosition.currentValue) {
            globalBestPosition.currentValue = currentPosition;
            //We want to copy the decision variables of the particle to the global best position,
                this way we can use it for the velocity value
            memcpy(globalBestPosition.decisionVariables, particleArray[i]->decisionVariables,
                NUM_VARIABLES * sizeof(double));
        }
    }
```

The commenting in the code provides a more detailed look into the logic behind this portion of the function however essentially it loops through all of the dimensions of the decision variables and the velocities and assigns random numbers to them within the bounds specified. It is worth noting that earlier in the function all of these lists were **malloced** so that the memory was properly accounted for to store the variables.

3.3 PSO Algorithmn

Most of the PSO algorithm follows the general structure that is outlined in the pseudocode at the bottom of the Assignment 4 PDF, however there is one key change that I would like to highlight in the report so that it can properly be explained. One of the key changes I made to allow for convergence of the high dimension, high complexity functions was to check for stagnant iterations. It is useful to note which functions are being referrenced when discussing high complexity function, these are the four functions which provided the most trouble when converging and required the extra modification to the algorithm:

1. Schwefel
2. Dixon-Price
3. Michalewicz
4. Styblinski-Tang

Ultimately I was unable to get Dixon-Price to converge for 50 variables, and I was unable to get both Schwefel and Dixon-Price to converge for 100 variables. It is likely that given more time and using a massive amount of particles that these would've eventually converged, especially with the stagnant particle managment.
With regards to the stagnant particle management, the following is how that was dealt with:

```
    if (particleArray[i]->stagnantIterations > STAGNANT_ITERATIONS) {

                //Loop through all of the dimensions of the particle
                for (int j = 0; j < NUM_VARIABLES; j++) {

                    //Move the particle in a random direction by a small amount
                    particleArray[i]->decisionVariables[j] += random_double(-0.8, 0.8);

                    //We need to clamp the position within bounds
                    particleArray[i]->decisionVariables[j] = fmax(bounds->lowerBound, fmin(bounds->
                        upperBound, particleArray[i]->decisionVariables[j]));

                    //reset the velocity
                    particleArray[i]->velocityVariables[j] = random_double(-1, 1);
                }
                particleArray[i]->stagnantIterations = 0;
            }
```

On every iterion of the algorithm, for each particle, if the best position of the particle does not iunmprove then we increment the stagnant iterations count for that particle by 1. If the stagnant iterations count for the particle reaches the cutoff value, which has been set to 50, however this can be adjusted, then the position of each of the decision variables will be stepped in a random direction by a random value within the bounds of the function. The position is clamped so that it doesn't go out of bounds. Additionally, the velocity is reset so that it can move in a new dircetion if needed.
The implementation of this function drastically increases the convergence of the algorithm, especially for higher dimensions.

4 Results

Below is the tabulated list of results for the functions. It is worth noting since we seed the random number generator each iteration with the current time, these results will vary in between runs, however eventually it will convergence if it has done so in the table below.

From this data we can see that, even with the large number of dimensions for some of the functions, the convergence occurs relatively fast. Comparing this to before the stagnant particle management was implemented it provides anywhere from a **2x** to a **10x** performace uplift.

The final code that needed to be developed for this project was the CPU timing. This was done using the global clock with the following function:

```
    clock_t start, end;
    double cpu_time_used;

```

Table 1: Num Vars = 10

Function	Lower Bound	Upper Bound	Particles	Iterations	Optimal Fitness	CPU Time (sec)
Griewank	-600	600	20000	1000	0.000000	1.164s
Levy	-10	10	10	1000	0.000000	0.003417s
Rastrigin	-5.12	5.12	2500	1000	0.000000	0.372904s
Rosenbrock	-5	10	3000	1000	0.000009	0.5465952s
Schwefel	-500	500	40000	1000	0.000127	7.328s
Dixon-Price	-10	1	20000	1000	0.000000	0.4259s
Michalewicz	0	π	1000	900	-9.6601517	0.418091s
Styblinski-Tang	-5	5	1000	10000	-391.661657	3.605633s

Table 2: Num Vars = 50

Function	Lower Bound	Upper Bound	Particles	Iterations	Optimal Fitness	CPU Time (sec)
Griewank	-600	600	50000	1000	0.000000	1.752017s
Levy	-10	10	10000	1000	0.000000	6.4723s
Rastrigin	-5.12	5.12	5000	52100	0.000000	238.8735s
Rosenbrock	-5	10	5000	19000	0.000000	82.5768s
Schwefel	-500	500	500	100000	0.000637	50.393102s
Dixon-Price	-10	10	1000	(missing)	(missing)	(missing)
Michalewicz	0	π	1000	10000	-46.084809	22.30346s
Styblinski-Tang	-5	5	1000	10000	-1958.308285	16.828267s

Table 3: Num Vars = 100

Function	Lower Bound	Upper Bound	Particles	Iterations	Optimal Fitness	CPU Time (sec)
Griewank	-600	600	7000	400	0.000000	4.033s
Levy	-10	10	7000	1000	0.000000	32.788439s
Rastrigin	-5.12	5.12	7000	24200	0.000000	564.275483s
Rosenbrock	-5	10	7000	23900	0.000000	303.136329s
Schwefel	-500	500	(missing)	1000	(missing)	(missing)
Dixon-Price	-10	10	(missing)	1000	(missing)	(missing)
Michalewicz	0	π	1000	10000	-88.831491	46.021296s
Styblinski-Tang	-5	5	1000	10000	-3916.61657	34.2172s

```c
    start = clock();

    double best_fitness = pso(objective_function, NUM_VARIABLES, bounds, NUM_PARTICLES,
        MAX_ITERATIONS, best_position);

    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    printf("CPU time used: %lf seconds\n", cpu_time_used);
```

In the next section of the report I will detail how to operate the code.

5 Instructions for Operation

Operation of the code is not challenging. Included with the code is a Makefile that allows the user to type **make** into the terminal and compile and link the code. Once the code is compiled all that is required from the user is to type **./main** followed by the parameters you want, practically this looks like **./main ObjectiveFunctionName NUM-VARIABLES LowerBound UpperBound NUM-PARTICLES MAX-ITERATIONS**.

A C Code

```c
//Include all the libraries that we need
#include "utility.h"
#include "OF_lib.h"
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <math.h>

//Define all of the constants that we need
#define INERTIAL_WEIGHT 0.7
#define COGNITIVE_COEFFICIENT 1.5
#define SOCIAL_COEFFICIENT 1.5
#define STAGNANT_ITERATIONS 100

// New random function because the old one wouldn't work for me, this uses drand48
double random_double(double min, double max) {

    //Create a random fraction from 0 to 1
    double random_fraction = drand48();

    //Return the random number that is generated within the bounds
    return min+ (max-min)*random_fraction;

}

//This is the main function that will be called by the user to run the PSO algorithm
double pso(ObjectiveFunction objective_function, int NUM_VARIABLES, Bound *bounds, int
    NUM_PARTICLES, int MAX_ITERATIONS, double *best_position) {

    //Create an array of particles
    Particle* particleArray[NUM_PARTICLES];

    //Malloc the memory for the particles
    for (int i = 0; i < NUM_PARTICLES; i++) {
        particleArray[i] = (Particle *)malloc(sizeof(Particle));

        //Check to see if the memory was allocated properly
        if (particleArray[i] == NULL) {
            printf("Error␣allocating␣memory␣for␣Particles...");
            exit(EXIT_FAILURE);
        }
    }

    //Create a global best position
    BestPosition globalBestPosition;

    // Initialize global best position
    globalBestPosition.decisionVariables = (double *)malloc(NUM_VARIABLES * sizeof(double));

    //Check to see if the memory was allocated properly
    if (globalBestPosition.decisionVariables == NULL) {
        printf("Error␣allocating␣memory␣for␣global␣best␣position...");
        exit(EXIT_FAILURE);
    }

    //Set the global best position to infinity, this is needed to ensure that the first particle
        will always be better
    globalBestPosition.currentValue = INFINITY;

    //Initialize all of the dimensions for each particle
    for(int i = 0; i < NUM_PARTICLES; i++) {


        //Malloc everything
        particleArray[i]->decisionVariables = (double *)malloc(NUM_VARIABLES*sizeof(double));
        particleArray[i]->personalBest = (double *)malloc(NUM_VARIABLES*sizeof(double));
        particleArray[i]->velocityVariables = (double *)malloc(NUM_VARIABLES*sizeof(double));

        //Iterate through all of the dimensions and set the values to random numbers
        for(int j=0; j < NUM_VARIABLES; j++) {
            double random_num = random_double(bounds->lowerBound, bounds->upperBound);

            //We want to set the decision variables and the personal best to the same value, that way we can iterate on them
            particleArray[i]->decisionVariables[j] = random_num;
            particleArray[i]->personalBest[j] = random_num;

            //We set the velocity to a random number between -1 and 1
            particleArray[i]->velocityVariables[j] = random_double(-1, 1);
        }


        //Calculate the current position of the particle and set it as both the current position and the personal best fitness
        //We do this because the first iteration will always be the best
        double currentPosition = objective_function(NUM_VARIABLES, particleArray[i]->decisionVariables);
        particleArray[i]->currentPosition = currentPosition;
        particleArray[i]->personalBestFitness = currentPosition;

        //Keep track of the global best position from the initialized particles
        if (currentPosition < globalBestPosition.currentValue) {
            globalBestPosition.currentValue = currentPosition;
            //We want to copy the decision variables of the particle to the global best position, this way we can use it for the velocity value
            memcpy(globalBestPosition.decisionVariables, particleArray[i]->decisionVariables, NUM_VARIABLES * sizeof(double));
        }
    }

    //Keep track of the while loop iterations
    int iterationCount = 0;

    //This is the number of convergence iterations
    while (iterationCount < MAX_ITERATIONS) {

        srand48(time(NULL));

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

            // Update velocity and position of the particles for each iteration
            //We need to loop through the variables for position and velocity of each partilce
            for (int j = 0; j < NUM_VARIABLES; j++) {

                //Generate two random number which will add some randomness to the velocity calculation
                double r1 = random_double(0, 1);
                double r2 = random_double(0, 1);

                // This is the velocity calculation formula which is done for each dimension
                //It is the one from the assignment document
                particleArray[i]->velocityVariables[j] = INERTIAL_WEIGHT* particleArray[i]->velocityVariables[j] +
                    COGNITIVE_COEFFICIENT * r1 * (particleArray[i]->personalBest[j] - particleArray[i]->decisionVariables[j]) +
                    SOCIAL_COEFFICIENT * r2 * (globalBestPosition.decisionVariables[j] - particleArray[i]->decisionVariables[j]);

                // Update position
                particleArray[i]->decisionVariables[j] += particleArray[i]->velocityVariables[j];

                // Clamp position within bounds
                particleArray[i]->decisionVariables[j] = fmax(bounds->lowerBound, fmin(bounds->upperBound, particleArray[i]->decisionVariables[j]));

                //Check to see if the position is within bounds, this will probably never happen due to the clamp but just in case
                if (particleArray[i]->decisionVariables[j] < bounds->lowerBound || particleArray[i]->decisionVariables[j] > bounds->upperBound) {
                    printf("Error:␣Particle␣%d,␣Dimension␣%d␣is␣out␣of␣bounds\n", i, j);
                    exit(EXIT_FAILURE);
                }
            }

            //Calculate the fitness of the current particle
            double position = objective_function(NUM_VARIABLES, particleArray[i]->decisionVariables);

            //Update current position of the particle
            particleArray[i]->currentPosition = position;

            // Update personal best if current position is better
            if (position < particleArray[i]->personalBestFitness) {

                //Copy the decision variables to the personal best
                memcpy(particleArray[i]->personalBest, particleArray[i]->decisionVariables, NUM_VARIABLES * sizeof(double));

                //Set the personal best fitness to the current position
                particleArray[i]->personalBestFitness = position;

                //Reset the stagnant iterations
                particleArray[i]->stagnantIterations = 0;
            }

            //If the current position is greater than or equal to the current best then we increment the stagnant iterations
            else {
                particleArray[i]->stagnantIterations += 1;
            }

            // Update global best if current position is better
            if (position < globalBestPosition.currentValue) {

                //Set the global best position to the current position
                globalBestPosition.currentValue = position;

                //Copy the decision variables to the global best position
                memcpy(globalBestPosition.decisionVariables, particleArray[i]->decisionVariables, NUM_VARIABLES * sizeof(double));
            }

            //This is the main difference between the pseudo code in the assignment document and the actual code
            //If the maximum number of stagnant iterations is reached, then we move the particle in a random direction by a small amount

            if (particleArray[i]->stagnantIterations > STAGNANT_ITERATIONS) {

                //Loop through all of the dimensions of the particle
                for (int j = 0; j < NUM_VARIABLES; j++) {

                    //Move the particle in a random direction by a small amount
                    particleArray[i]->decisionVariables[j] += random_double(-0.8, 0.8);

                    //We need to clamp the position within bounds
                    particleArray[i]->decisionVariables[j] = fmax(bounds->lowerBound, fmin(bounds->upperBound, particleArray[i]->decisionVariables[j]));

                    //reset the velocity
                    particleArray[i]->velocityVariables[j] = random_double(-1, 1);
                }
                particleArray[i]->stagnantIterations = 0;
            }
        }

        // Print fitness every 100 iterations
        //I have commented this out but you can turn it back on if you want to see the fitness
            every 100 iterations
        /*
        if (iterationCount % 100 == 0) {
            printf("Iteration %d: Best Fitness = %.30f\n", iterationCount, globalBestPosition.
                currentValue);

            //print the first two particles current fitness positions for debugging

            printf("Particle 0: %.30f\n", particleArray[0]->currentPosition);
            printf("Particle 1: %.30f\n", particleArray[1]->currentPosition);
        }
        */

        // Optional stopping condition, this can be turned on and off and only acts as a stopping
            condition but fucntions that converge to 0
        if ((globalBestPosition.currentValue < 1e-4 && globalBestPosition.currentValue > 0)) {
            break;
        }

        // Increment iteration count
        iterationCount += 1;
    }

    // Copy global best position to output parameter
    memcpy(best_position, globalBestPosition.decisionVariables, NUM_VARIABLES * sizeof(double));

    // Free allocated memory
    for (int i = 0; i < NUM_PARTICLES; i++) {
        free(particleArray[i]->decisionVariables);
        free(particleArray[i]->personalBest);
        free(particleArray[i]->velocityVariables);
        free(particleArray[i]);
    }
    free(globalBestPosition.decisionVariables);

    //Return the global best position

    return globalBestPosition.currentValue;
}

/*
int main(void) {

    //Create a seed for the random number generation
    srand48(time(NULL));

    return 0;
}
*/
```

Listing 1: C code for the PSO algorithm implementation.

```c
#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;

//Define the struct for the particle
//The struct is used to help with readibility of the program and to make designing the logic much
    easier
typedef struct Particle {
    //Array for the dimensions of the particle
    double *decisionVariables;
    //Array for the personal best position of the particle
    double *personalBest;
    //Array for the velocity of the particle
    double *velocityVariables;
    //The current position of the particle
    double currentPosition;
    //The personal best fitness of the particle
    double personalBestFitness;
    //The number of stagnant iterations
    int stagnantIterations;
} Particle;

//This could've been a particle but to save memory, better to have its own type
typedef struct BestPosition {
    double *decisionVariables;
    double currentValue;
} BestPosition;


// 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[]);

#endif // UTILITY_H
```

Listing 2: C code header for the function defitions and the package headers