MECHTRON 2MP3

Developing a Genetic Optimization Algorithm in C

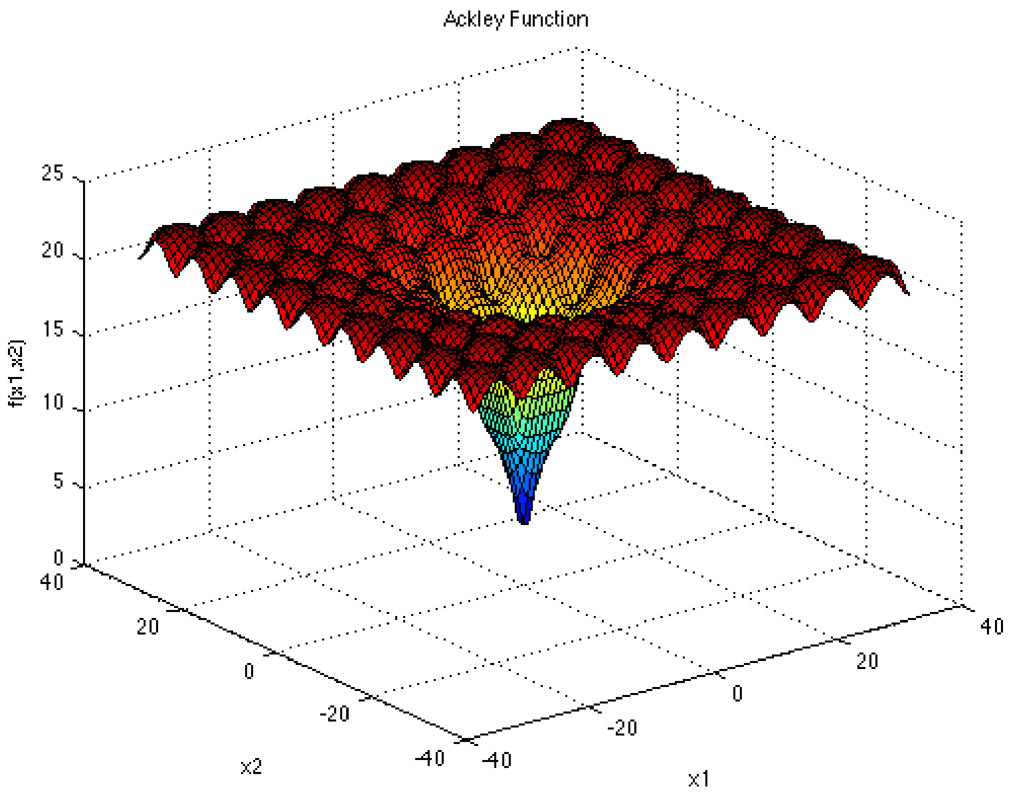
Grant McClure

1 Introduction

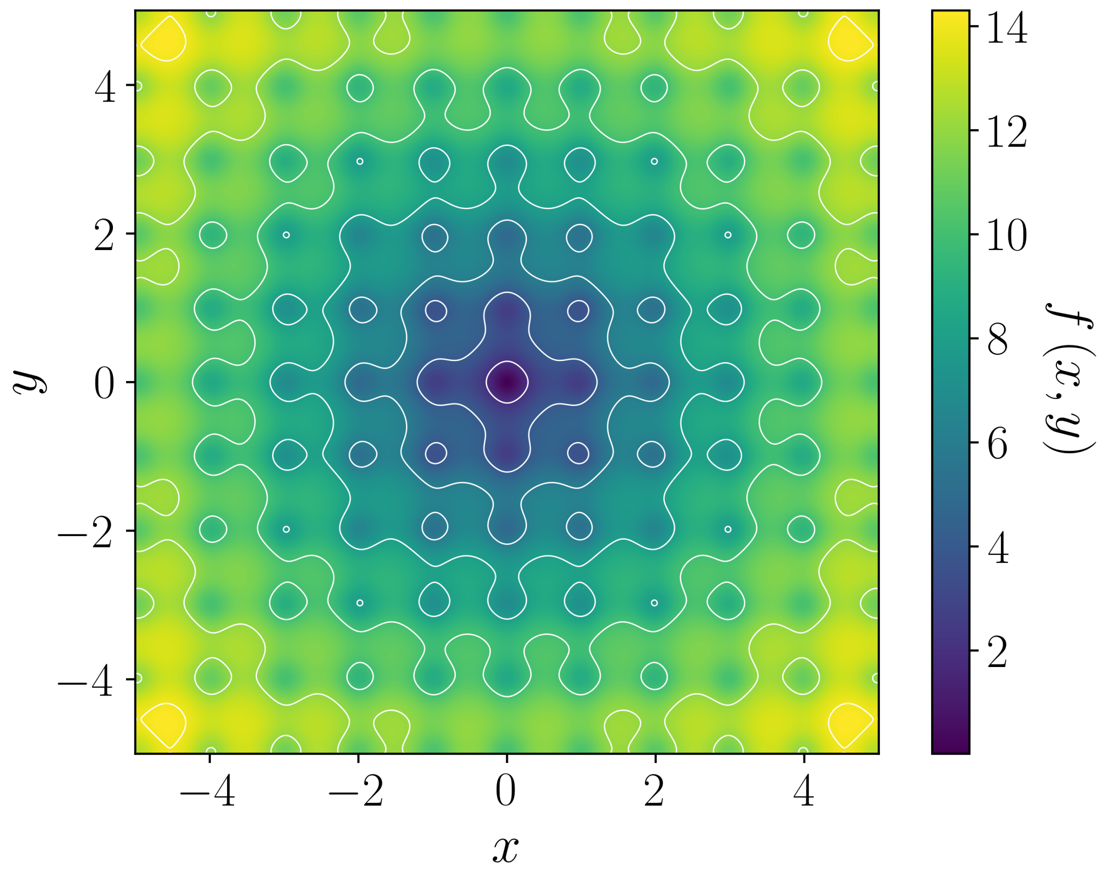
The objective of this assignment was to minimise the Ackley function using a genetic algorithm. The Ackley function referred to as the objective function shown below:

$$f(x) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

The visual representation of the Ackley Function is shown below:



As shown in the graph above it can be seen that the ackely function has lots of local minima, which makes it difficult to solve for global minima, looking at the contour plot of the same function:



The contour plot shown above show that the global minima which is being solved for is at roughly (0,0), therefore the aim for this genetic algorithm is to solve for solutions as close to the global minima.

2 Programming Genetic Algorithm

2.1 Implementation

The number of decision variables in the optimization problem is set to 2, with upper and lower bounds for both decision variables set to +5 and -5, respectively:

The code written contained five main functions, contained in `functions.c`:

- Generate Random Number between a minimum and maximum
- Generate a random unsigned integer
- generate random population with each individual having two alleles (two x values)

- Crossover - Chooses random individuals within population and creates two parents, these parents then cross-over one of their alleles two create a new individual. The chance of this happening is controlled by the *crossover rate* variable. This is one-point crossover.
- Mutation - Selects random individuals with population and generates a random number between the bounds, and swaps this with the current individuals alleles, this introduces new alleles into the new population, in order for higher chance for these to be selected for. This adds more selection pressure that can be controlled.

Within `GA.c` there are more functions required to execute the Genetic Algorithm these will be discussed in 2.3. The following results were obtained, when the `GA.c`, was run with the follwing parameters.

Table 1: Results with Crossover Rate = 0.5 and Mutation Rate = 0.05

Pop Size	Max Gen	Best Solution			CPU time (Sec)
		x_1	x_2	Fitness	
10	100	-0.1044163411	-0.0893913559	0.8342040320	0.026170
100	100	-0.0563895307	-0.0039102067	0.2427591449	0.095326
1000	100	-0.0014466210	-0.0000768318	0.0041533107	0.228272
10000	100	-0.0015135412	-0.0089275907	0.0002779324	12.393531
1000	1000	0.0000365754	-0.0006927061	0.0019748109	1.638280
1000	10000	0.0000191503	-0.0000420050	0.0001306294	17.119482
1000	100000	0.0000506896	0.0000167568	0.0001510785	306.195726
1000	1000000	-0.0010140869	0.0004970119	0.0032281975	429.695955

Table 2: Results with Crossover Rate = 0.5 and Mutation Rate = 0.2

Pop Size	Max Gen	Best Solution			CPU time (Sec)
		x_1	x_2	Fitness	
10	100	0.0043231365	0.0437927502	0.1752221205	0.004628
100	100	-0.0047118613	0.0036475784	0.0177990891	0.016496
1000	100	0.0003357721	-0.0060206768	0.0180234605	0.190825
10000	100	-0.0015232689	-0.0001465087	0.0043906947	4.641066
1000	1000	-0.0001575961	0.0004610536	0.0013844565	1.273733
1000	10000	0.0001237378	0.0000705919	0.0004034724	14.863587
1000	100000	-0.0002094288	0.0000181072	0.0005957408	145.914189
1000	1000000	0.0001565716	0.0004869723	0.00014537755	611.262866

2.2 Result Analysis and `Makefile`

The following conclusions can be made on the data from 2.1:

1. **Population Size** - As populations size increases, there is a clear trend of increasing fitness values. This is due to there being a wider genetic pool therefore the algorithm can sample the space much more efficiently. Higher chance of converging towards global minimum

2. **Generations and Convergence** - It can be seen that there is a trend, between increasing the number of generations and the value of fitness.

$$Generations \ \alpha \ \frac{1}{fitness}$$

   Improvement of fitness is due to that the mutation and crossover scope have more range to work over.

3. **Computational Time** It can be seen that as the number of required calculations increase, the computation time increases dramatically, especially when there is a high population as well as high number of generations. However, it can be said that there must be a sweet spot where the balance between computational time and solution quality is optimal. This is somewhere between 10,000 to 1,000,000 generations. As in this range there is marginal difference between the quality of fitness values.

2.2.1 Makefile

Below is the code for the Makefile used to compile the C code used in the genetic algorithm, and what each line does.

```
# Define the compiler - gcc
CC = gcc

# Compiler flags

CFLAGS = -Wall -g -pg -lm

# Source files
SOURCES = GA.c functions.c OF.c

# Include directories
INCLUDES = -I.

# Libraries
LIBS = -lm

# Output executable
```

```
TARGET = GA

# Default rule
all: $(TARGET)

# Compile the source files and link them to create the
   executable
$(TARGET): $(SOURCES)
 $(CC) $(CFLAGS) $(INCLUDES) -o $(TARGET) $(SOURCES) $(LIBS)

# Clean up object files and the executable
clean:
 rm -f $(TARGET)

# PHONY rule to avoid conflicts with filenames
.PHONY: clean
```

When compiling the following flags are used: -Wall enables all compiler's warning messages. -g enables debugging information in the executable. -pg enables profiling information to be included, useful for performance analysis. -lm links the math library

2.3 Improving the Performance

In order to increase performance the following functions were used:

2.3.1 Elitism

Elitism is used to ensure the fittest individuals/best solutions are preserved and don't undergo crossover and/or mutation, the alleles are passed directly onto the next generation.

The code for Elitism is shown below:

```
    int elite_index = 0;                 // find index of most elite
        individual
        double elite_fitness = __DBL_MAX__; // set to super high
           indivual

        for (int i = 0; i < POPULATION_SIZE; i++)
        {
            if (fitness[i] < elite_fitness)
            {
                elite_fitness = fitness[i]; // find indi wih best
                   fitness
                elite_index = i;
            }
        }

        // keep best indiviaial - store
        double elite_individual[NUM_VARIABLES];
        for (int i = 0; i < NUM_VARIABLES; i++)
        {
            elite_individual[i] = population[elite_index][i];
        }
```

Then after mutation and crossover, the best individual is passed directly to the next population:

```
    for (int i = 0; i < NUM_VARIABLES; i++)
        {
            population[0][i] = elite_individual[i]; // replace
               first individial with most elite
        }
```

When elitism is used it causes the populations to converge faster by retaining the best individuals, elitism can help the algorithm to converge more quickly towards an optimal or near-optimal solution, as there is a continual improvement in the population's fitness over generations. As this is a genetic algorithm it is based on what happens in nature, Elitism is natural selection where the fittest individuals are allowed to pass on their genes.

Elitism can also ensure that diversity is maintained within the population by moving the best individuals into the next population prevents the current population from becoming too homogeneous leading to premature convergence or non-optimal solutions like finding a local minima over the global minima. When writing the algorithm this was one of the challenges that had to be overcome, as the algorithm initially kept finding local minima this was fixed by incorporating Elitism.

The plots comparing the fittest individuals over generations, with and without elitism are shown below, as well as the contour plots of the final populations with and without elitism:

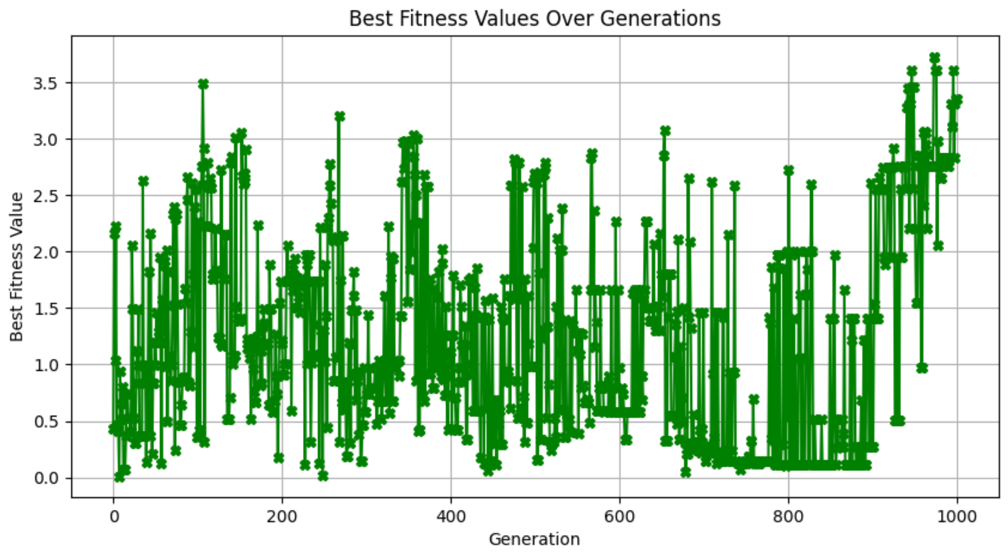


Figure 1: Fitness over generations without elitism.

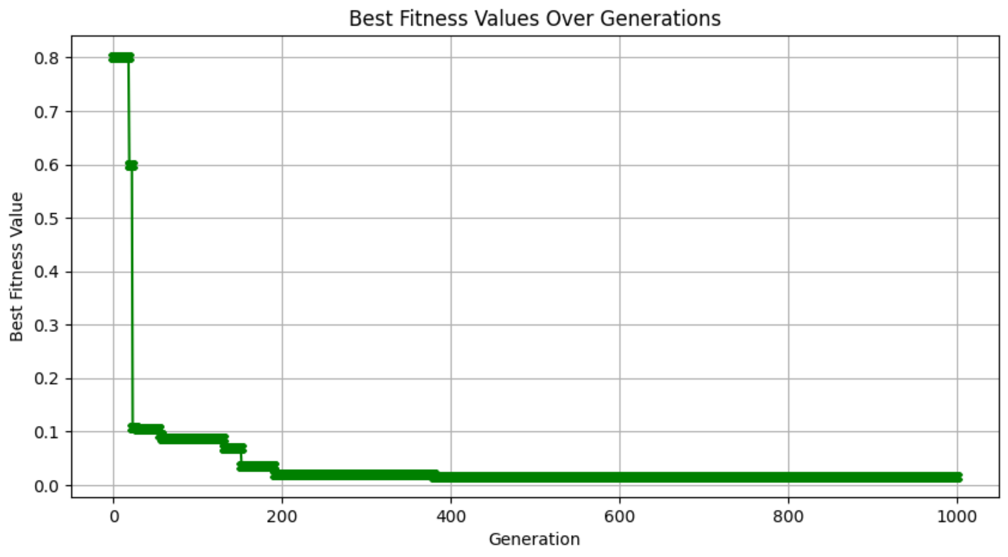


Figure 2: Fitness over generations with elitism.

The contour plots below show there is little to no difference in the location of the individuals within the solution due to Elitism

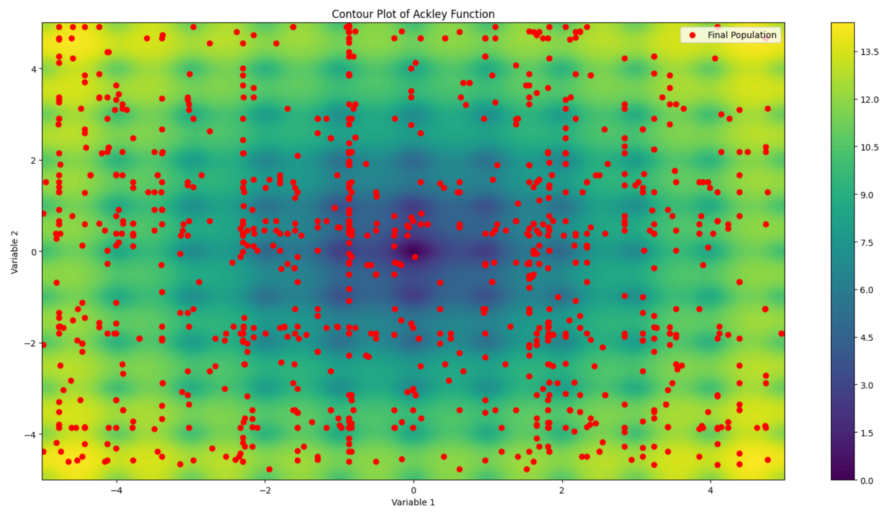


Figure 3: Population over generations without elitism.

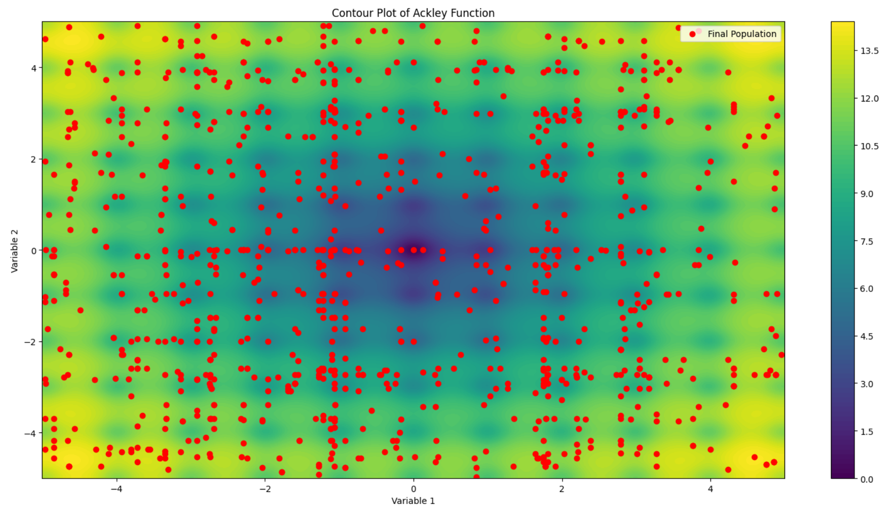


Figure 4: Population over generations with elitism.

2.3.2 Adaptive Mutation and Crossover

By using adaptive methods in Genetic algorithms, it can help prevent premature convergence, by increasing/decreasing mutation and crossover rates, based on how diverse the population is, this can introduce new genetic information when required.

These methods allow more exploration of the solution space, while ensuring that the algorithm doesn't get stuck in local minima as mutation rate will be increased, which will prevent convergence, within anywhere outwith global minima, as mutation rate will slow down as fitness improves.

The code for adaptive mutation and crossover rate is shown below:

```
//set mutation rates variables to start and end with
     mutate_rate = adaptiveMutationRate(generation, MAX_GENERATIONS
        , initial_mutate_rate, final_mutate_rate);
        crossover_rate = adaptiveCrossoverRate(generation,
           MAX_GENERATIONS, initial_crossover_rate,
           final_crossover_rate);

        double adaptiveMutationRate(int generation, int
           MAX_GENERATIONS, double startRate, double endRate)
{
    //Linearly decrease the mutation rate from startRate to endRate
        over the generations
    double rate = startRate + (endRate - startRate) * ((double)
       generation / (double)MAX_GENERATIONS);
    return rate < endRate ? endRate : rate; // Ensure that the rate
        never goes below the endRate
}

double adaptiveCrossoverRate(int generation, int MAX_GENERATIONS,
   double startRate, double endRate)
{
    //Linearly decrease the crossover rate from startRate to
       endRate over the generations
    double rate = startRate + (endRate - startRate) * ((double)
       generation / (double)MAX_GENERATIONS);
    return rate < endRate ? endRate : rate; // ensure that the rate
        never goes below the endRate
}
```

When the fitness is plotted with and without adaptive behaviour the following results are observed:

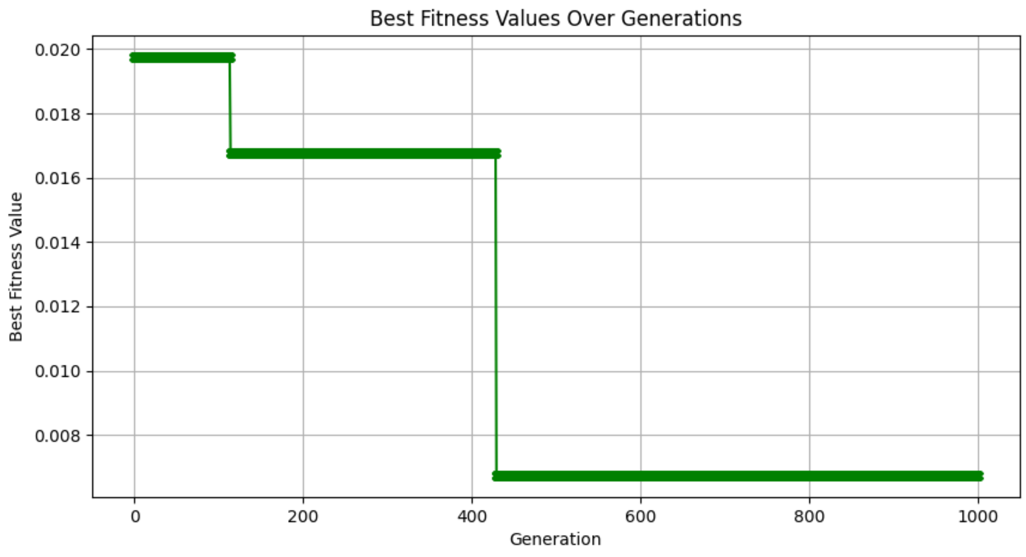


Figure 5: Fitness over generations without Adaptive.

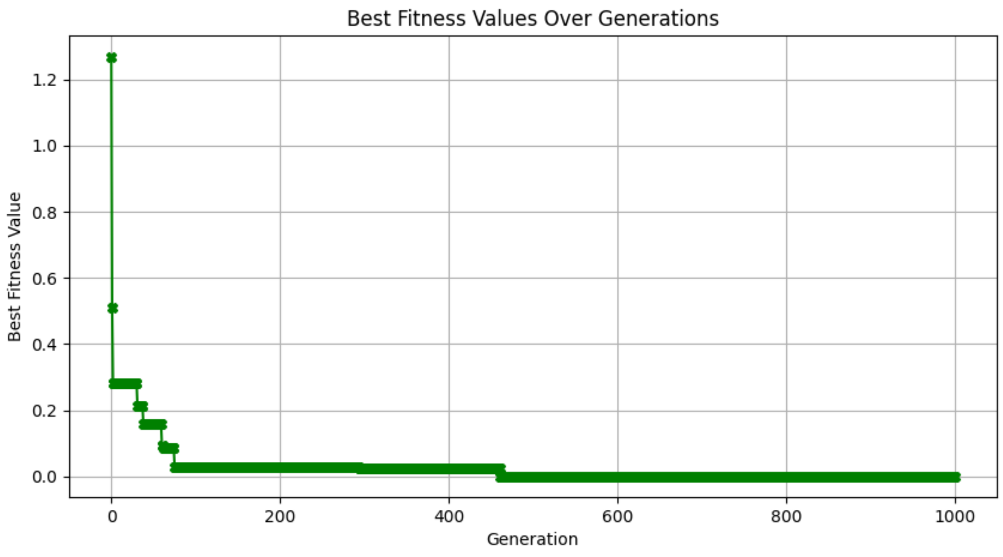


Figure 6: Fitness over generations with Adaptive.

In the figures above it is seen that with adaptive crossover and mutation rates, the fitness values converge much quicker as once near global minima mutation rate decreases in order to find the most optimum solution.

When the populations are plotted with and without adaptive mutation and crossover rate, the populations with adaptive behavior, converge around minima of the function, this allows better searching of the space, and finds better solutions overall. This is shown below:

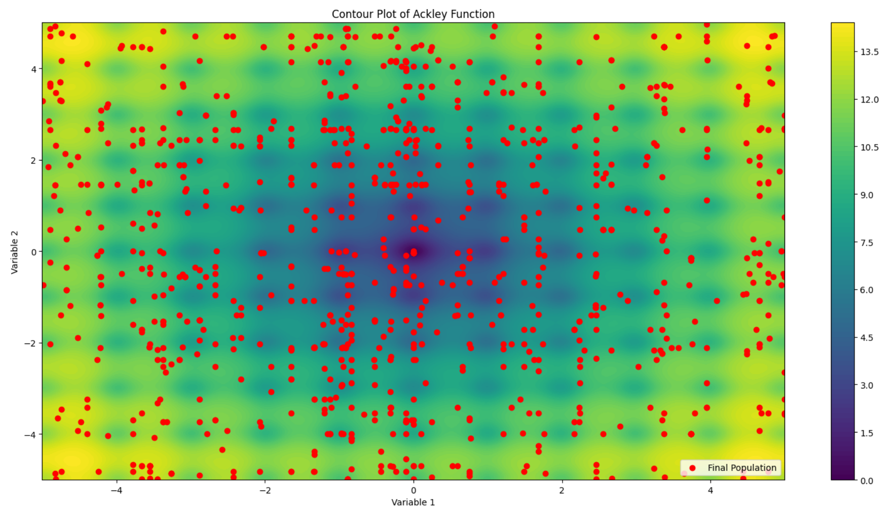


Figure 7: Population over generations without Adaptive.

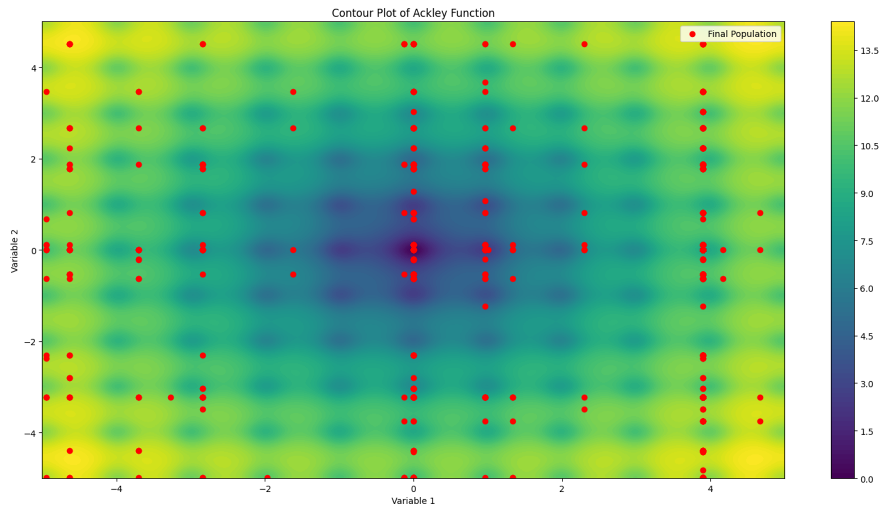


Figure 8: Population over generations with Adaptive.

2.4 Improving Speed

Introducing extra functions to the genetic algorithm, caused the computation time to decrease. This is shown in the following:

Running `gprof` by using the -pg flag and using the command:
`gprof GA gmon.out > analysis.txt`

```
    Each sample counts as 0.01 seconds.
```

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 96.75      1.19     1.19                             main
  1.63      1.21     0.02     1000    20.00    20.00  crossover
  0.81      1.22     0.01  1000000     0.01     0.01
     Objective_function
  0.81      1.23     0.01     1000    10.00    20.00
     compute_objective_function
  0.00      1.23     0.00  1052550     0.00     0.00
     generate_random
  0.00      1.23     0.00    50550     0.00     0.00  generate_int
  0.00      1.23     0.00     1000     0.00     0.00
     adaptiveCrossoverRate
  0.00      1.23     0.00     1000     0.00     0.00
     adaptiveMutationRate
  0.00      1.23     0.00     1000     0.00     0.00
     calculateDistance
  0.00      1.23     0.00     1000     0.00     0.00  mutate
  0.00      1.23     0.00     1000     0.00     0.00
     printProgressBar
  0.00      1.23     0.00        1     0.00     0.00
     generate_population
```

This shows that main function takes up 96.75% of the computation time, this is where the functions are called which shouldn't have an effect, however this is where plotting is stored and elitism calculation is done. To improve speed of code the main function needs to be more efficient with memory, using parallel computing and dynamic memory allocation would help.

3 Appendix

3.1 Python Plotting Code

```python
    import matplotlib.pyplot as plt

# Read best fitness values from the text file
with open("best_fitness_values.txt", "r") as file:
    best_fitness_values = [float(line) for line in file]

```

```
# Plotbest itness values
plt.figure(figsize=(10, 5))
plt.plot(best_fitness_values, marker='X', linestyle='-', color = 'g
   ')
plt.title("Best Fitness Values Over Generations")
plt.xlabel("Generation")
plt.ylabel("Best Fitness Value")
plt.grid(True)

#Savethe plot as an image or display it
plt.savefig("best_fitness_plot.png")
plt.show() ##show plot

print("Best fitness plot saved as 'best_fitness_plot.png'")
```

```
    import numpy as np
import matplotlib.pyplot as plt

# Define the Ackley function
def ackley(x, y):
    a = -20 * np.exp(-0.2 * np.sqrt(0.5 * (x**2 + y**2)))
    b = -np.exp(0.5 * (np.cos(2 * np.pi * x) + np.cos(2 * np.pi * y
       )))
    return a + b + 20 + np.exp(1)

# Create a mesh grid for the x and y coordinates
x = np.linspace(-5, 5, 400)
y = np.linspace(-5, 5, 400)
X, Y = np.meshgrid(x, y)

# Calculate the Ackley function values on the mesh grid
Z = ackley(X, Y)

# Create a contour plot of the Ackley function
plt.contourf(X, Y, Z, levels=100, cmap='viridis')
plt.colorbar()
plt.title("Contour Plot of Ackley Function")

#plot final pop
```

```python
final_population = np.loadtxt("final_population.txt")
variable1_values = final_population[:, 0]
variable2_values = final_population[:, 1]
plt.scatter(variable1_values, variable2_values, c='red', marker='o'
    , label="Final Population")

plt.xlabel('Variable 1')
plt.ylabel('Variable 2')
plt.legend()

plt.show()
```

3.2 Objective Function - `OF.c`

```c
#include <stdio.h>
#include <math.h>

// Do know change anything in this file. You will not submit this
    file.

// Define the objective function function for optimization
double Objective_function(int NUM_VARIABLES, double x[NUM_VARIABLES
    ])
{

    // While the objective function could be anything here it is to
        minimize Ackley function
    double sum1 = 0.0;
    double sum2 = 0.0;
    for (int i = 0; i < NUM_VARIABLES; i++)
    {
        sum1 += x[i] * x[i];
        sum2 += cos(2.0 * M_PI * x[i]);
    }
    return -20.0 * exp(-0.2 * sqrt(sum1 / NUM_VARIABLES)) - exp(
       sum2 / NUM_VARIABLES) + 20.0 + M_E;
}
```

3.3 Functions.c

```
    // Include everything necessary here
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <limits.h>
#include <math.h>

extern double Objective_function(int NUM_VARIABLES, double x[
   NUM_VARIABLES]);  //Import OF

double generate_random(double min, double max)
{
    if (max < min)
    {
        printf("Max cannot be less than the minimum value\n");
        exit(1);
    }

    double random_number = (double)rand() / RAND_MAX;    //
       Generate a random value between 0 and 1
    random_number = (random_number * (max - min)) + min; //scale to
        in rangee


    return random_number;
}

unsigned int generate_int()
{
    unsigned int random_int = rand();

    if (random_int > UINT_MAX || random_int < 0) //double check
       correct size
    {
        printf("Not an unsigned integer\n"); //checl error
        exit(1);
    }
    return random_int;
}

void generate_population(int POPULATION_SIZE, int NUM_VARIABLES,
   double population[POPULATION_SIZE][NUM_VARIABLES], double Lbound
   [NUM_VARIABLES], double Ubound[NUM_VARIABLES])
{
    for (int i = 0; i < POPULATION_SIZE; i++)
    {
        for (int j = 0; j < NUM_VARIABLES; j++)
        {
            population[i][j] = generate_random(Lbound[j], Ubound[j
               ]); //random generic population within bounds
        }
    }
}
//call objecive fucntion
void compute_objective_function(int POPULATION_SIZE, int
   NUM_VARIABLES, double population[POPULATION_SIZE][NUM_VARIABLES
   ], double fitness[POPULATION_SIZE])
{
    for (int i = 0; i < POPULATION_SIZE; i++)
    {
        // Call the Objective_function from OF.c with the
           appropriate arguments
        fitness[i] = Objective_function(NUM_VARIABLES, population[i
           ]); //computes fitness and stores in array
    }



}

void crossover(int POPULATION_SIZE, int NUM_VARIABLES, double
   fitness[POPULATION_SIZE], double new_population[POPULATION_SIZE
   ][NUM_VARIABLES], double population[POPULATION_SIZE][
   NUM_VARIABLES], double crossover_rate)
{
   int selected_indices[POPULATION_SIZE]; //create an array of
      selected for indices
    // Select random indices for crossover
    for (int i = 0; i < POPULATION_SIZE; i++) {
        selected_indices[i] = rand() % POPULATION_SIZE;
    }

    // Perform crossover
    for (int i = 0; i < POPULATION_SIZE; i += 2) {
        if ((double)rand() / RAND_MAX < crossover_rate) {
            int parent1_index = selected_indices[i]; //create
               parent1 index
            int parent2_index = selected_indices[i + 1]; //create
               oarent 2 index
            int crosspoint = rand() % (NUM_VARIABLES - 1) + 1; //
               Ensure crosspoint is  not at end of array

            // create new individuals using one point crossover
            for (int j = 0; j < crosspoint; j++) {
                new_population[i][j] = population[parent1_index][j
                   ]; //mix genes
                new_population[i + 1][j] = population[parent2_index
                   ][j]; //mix genes
            }
            for (int j = crosspoint; j < NUM_VARIABLES; j++) {
                new_population[i][j] = population[parent2_index][j
                   ]; //mix other gene
                new_population[i + 1][j] = population[parent1_index
                   ][j];//miz other genes after crosspoint
            }
        } else {
            // If crossover doesn't happen, copy the parents to the
                new population
            for (int j = 0; j < NUM_VARIABLES; j++) {
                new_population[i][j] = population[selected_indices[
                   i]][j];
                new_population[i + 1][j] = population[
                   selected_indices[i + 1]][j];
            }
        }
    }
}

void mutate(int POPULATION_SIZE, int NUM_VARIABLES, double
   new_population[POPULATION_SIZE][NUM_VARIABLES], double
   population[POPULATION_SIZE][NUM_VARIABLES], double Lbound[
   NUM_VARIABLES], double Ubound[NUM_VARIABLES], double mutate_rate
   )
{


    //this is directly translated from the python mutate code
       provided

    //calcluate number of genes in popluation
    int total_genes = POPULATION_SIZE * NUM_VARIABLES;

    // calculate mutate rate
    int genes_to_mutate = (int)(total_genes * mutate_rate);

    // Perform mutations
    for (int i = 0; i < genes_to_mutate; i++)
    {
        //find index to mutate
        int gene_index = generate_int() % total_genes;

        // Calculate the row and column of the gene to mutate
        int row = gene_index / NUM_VARIABLES;
        int col = gene_index % NUM_VARIABLES;

        // Mutate the gene within the bounds
        new_population[row][col] = generate_random(Lbound[col],
           Ubound[col]);
    }
}
```

3.4 `GA.c`

```c
    #include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <stdbool.h>
#include "functions.h"

// function prototypes
void printProgressBar(int iteration, int total);
double calculateDistance(double x, double y);
double adaptiveCrossoverRate(int generation, int MAX_GENERATIONS,
   double startRate, double endRate);
double adaptiveMutationRate(int generation, int MAX_GENERATIONS,
   double startRate, double endRate);

int main(int argc, char *argv[])
{
    /*all storage variabels*/

    srand((unsigned int)time(NULL)); // seed rng
    int best_index = -1;
    double best_fitness = __DBL_MAX__;
    double prev_best_fitness = __DBL_MAX__;
    int closest_solution_index = -1;
    double global_best_fitness = __DBL_MAX__;
    int no_improvement_counter = 0;
    double previous_best_fitness = __DBL_MAX__;

    // <YOUR CODE: Handle the possible errors in input data given
       by the user and say how to execute the code>
    if (argc != 6) // check correct number of arguments
    {
        printf("Usage: %s POPULATION_SIZE MAX_GENERATIONS
           crossover_rate mutate_rate stop_criteria\n", argv[0]);
        printf("%d", argc);
        return 1;
    }
    // <YOUR CODE: Assign all inputs given by the user argv[i]>
       like

    // cconvert argymnets to coorenct date types
    unsigned int POPULATION_SIZE = atoi(argv[1]);
    unsigned int MAX_GENERATIONS = atoi(argv[2]);
    double crossover_rate = atof(argv[3]);
    double mutate_rate = atof(argv[4]);
    double stop_criteria = atof(argv[5]);

    const int NO_IMPROVEMENT_THRESHOLD = MAX_GENERATIONS/3 ; // set
        an arbirtay convergnece criteria

    // possive erors
    if (mutate_rate >= 1 || crossover_rate >= 1)
    {
        printf("incorrect mutate or crossover rate\n");
        exit(1);
    }
    if (POPULATION_SIZE > __UINT64_MAX__ || MAX_GENERATIONS >
       __UINT64_MAX__)
    {
        printf("population size or max generations too big too big\
          n");
        exit(1);
    }
    if(POPULATION_SIZE < 0 || MAX_GENERATIONS <0 || crossover_rate
       <0 || mutate_rate < 0 || stop_criteria< 0){
        printf("no negative input arguments\n");
        exit(1);
    }

    // POPULATION_SIZE, MAX_GENERATIONS, crossover_rate,
       mutate_rate, stop_criteria

    // ##############################################################
       #######################
    // you dont need to change anything here
    // the number of variables
    int NUM_VARIABLES = 2;
    // the lower bounds of variables
    double Lbound[] = {-5.0, -5.0};
    // the upper bounds of variable
    double Ubound[] = {5.0, 5.0};
    // ############################################################
       #######################

    // <YOUR CODE: Here make all the initial print outs>

    printf("genetic algorithm initiated\n");
    printf("====================================\n");
    printf("number of varibles: %d\n", NUM_VARIABLES);
    for (int i = 0; i < 1; i++)
    {
        printf("lower bound: %.10lf , %.10lf\n", Lbound[0], Lbound
           [1]);
    }

    for (int i = 0; i < 1; i++)
    {
        printf("Upper bound: %.10lf , %.10lf\n", Ubound[0], Ubound
           [1]);
    }

    printf("\n");

    printf("POPULATION_SIZE : %d\n", POPULATION_SIZE);
    printf("MAX GENERATIONS : %d\n", MAX_GENERATIONS);
    printf("crossover rate: %lf\n", crossover_rate);
    printf("mutate_rate : %lf\n", mutate_rate);
    printf("stop criteria : %.20lf\n", stop_criteria);

    printf("results\n");
    printf("====================================\n");

    clock_t start_time, end_time;
    double cpu_time_used;
    start_time = clock();
```

```
// <YOUR CODE: Declare all the arrays you need here>
double population[POPULATION_SIZE][NUM_VARIABLES];
double fitness[POPULATION_SIZE];
double new_population[POPULATION_SIZE][NUM_VARIABLES];
double best_fitness_values[MAX_GENERATIONS];
double final_population[POPULATION_SIZE][NUM_VARIABLES];

// <YOUR CODE: Call generate_population function to initialize
   the "population"> like:
generate_population(POPULATION_SIZE, NUM_VARIABLES, population,
    Lbound, Ubound);


double fitness_probs[POPULATION_SIZE];


/*mutation rates, are adaptive thorughout depending on where
   solution is foib*/
double initial_mutate_rate = mutate_rate; // these can chamge
   depending on scenatio
double final_mutate_rate = 0.001;          //

double initial_crossover_rate = crossover_rate;
double final_crossover_rate = 0.6;

/*issue loop only running once put in break points*/
for (int generation = 0; generation < MAX_GENERATIONS;
   generation++)
{

    //   <YOUR CODE: Compute the fitness values using objective
         function for
    //   each row in "population" (each set of variables)> like
       :
    //   compute_objective_function(POPULATION_SIZE,
       NUM_VARIABLES, population, fitness);

    // call functios
```

```
compute_objective_function(POPULATION_SIZE, NUM_VARIABLES,
   population, fitness);
mutate_rate = adaptiveMutationRate(generation,
   MAX_GENERATIONS, initial_mutate_rate, final_mutate_rate)
   ;
crossover_rate = adaptiveCrossoverRate(generation,
   MAX_GENERATIONS, initial_crossover_rate,
   final_crossover_rate);

/* Elitism - this section takes the genes o the fittest
   individual and pastes into the next generation */

int elite_index = 0;                // find index of most
   elite individual
double elite_fitness = __DBL_MAX__; // set to super high
   indivual - will change

for (int i = 0; i < POPULATION_SIZE; i++)
{
    if (fitness[i] < elite_fitness)
    {
        elite_fitness = fitness[i]; // find indi wih best
           fitness
        elite_index = i;
    }
}

// keep best indiviaial - store
double elite_individual[NUM_VARIABLES];
for (int i = 0; i < NUM_VARIABLES; i++)
{
    elite_individual[i] = population[elite_index][i];
}

// <YOUR CODE: Here implement the logic of finding best
   solution with minimum fitness value
// and the stopping criteria>
```

```
        /* code to calculate fintess probabilties affects how
           fitness is ppassed on*/

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

            if (fitness[i] < best_fitness)
            {
                best_fitness = fitness[i]; // find best i=fintess
                best_index = i;
            }
        }
        double sum_fitness_probs = 0;
        for (int i = 0; i < POPULATION_SIZE; i++)
        {
            fitness_probs[i] = 1 / (fitness[i] + 1e-5); // find i=
               fitness probs
            sum_fitness_probs += fitness_probs[i];
        }
        for (int i = 0; i < POPULATION_SIZE; i++)
        {
            fitness_probs[i] /= sum_fitness_probs;
        }

        // Selection of parents based on fitness probabilities and
           creation of a new population
        for (int i = 0; i < POPULATION_SIZE; i++)
        {

            double random_value = generate_random(0, 1); //
               Generate a random value between 0 and 1
            double cumulative_prob = 0.0;

            int selected_index = -1; // Initialize with an invalid
               value

            for (int j = 0; j < POPULATION_SIZE; j++)
            {
                cumulative_prob += fitness_probs[j];
                if (random_value <= cumulative_prob)
                {
                    selected_index = j;
                    break; // if seected index found
                }
            }

            if (selected_index != -1)
            {
                // Copy the selected individual to the new
                   population
                for (int k = 0; k < NUM_VARIABLES; k++)
                {
                    new_population[i][k] = population[
                       selected_index][k];
                }
            }
        }

        // <YOUR CODE: Here call the crossover function>
        crossover(POPULATION_SIZE, NUM_VARIABLES, fitness,
           new_population, population, crossover_rate);

        // <YOUR CODE: Here call the mutation function>
        mutate(POPULATION_SIZE, NUM_VARIABLES, new_population,
           population, Lbound, Ubound, mutate_rate);
        // recompite from loop
        for (int i = 0; i < POPULATION_SIZE; i++)
        {
            for (int j = 0; j < NUM_VARIABLES; j++)
            {
                population[i][j] = new_population[i][j]; // copy
                   new pop into pop (maybe unexxasy)
            }
        }

        for (int i = 0; i < NUM_VARIABLES; i++)
        {
            population[0][i] = elite_individual[i]; // replace
               first individial with most elite
        }

        prev_best_fitness = best_fitness;

        /*stop criteria*/

        double fitness_change = fabs(previous_best_fitness -
           best_fitness);
        if (best_fitness < global_best_fitness)
        {
            // There is an improvement
            global_best_fitness = best_fitness;
            previous_best_fitness = best_fitness; // Update
               previous best to current best
            no_improvement_counter = 0;           // Reset the
               counter since we found a better fitness
            // printf("found better solution\n");
        }
        else if (fitness_change < stop_criteria)
        {
            // No significant improvement
            no_improvement_counter++; // Increment the no
               improvement counter
            // printf("no improvment\n");
        }

        // Check if the no improvement threshold has been reached
        if (no_improvement_counter >= NO_IMPROVEMENT_THRESHOLD)
        {
            printf("Stopping: No signifcant improvment in best
               fitness for %d generations.\n",
               NO_IMPROVEMENT_THRESHOLD);
            break; // Exit the generation loop
        }

        // for plotting (done  by chatgpt)
        double generation_best_fitness = __DBL_MAX__;
        for (int i = 0; i < POPULATION_SIZE; i++)
        {
            if (fitness[i] < generation_best_fitness)
            {
                generation_best_fitness = fitness[i];
            }
        }
        if (generation == MAX_GENERATIONS - 1)
        {
            for (int i = 0; i < POPULATION_SIZE; i++)
            {
                for (int j = 0; j < NUM_VARIABLES; j++)
                {
                    final_population[i][j] = population[i][j];
                }
            }
        }

        // Update the array with the best fitness value for this
           generation
        best_fitness_values[generation] = generation_best_fitness;
        FILE *best_fitness_file = fopen("best_fitness_values.txt",
           "w");
        if (best_fitness_file)
        {
            for (int generation = 0; generation < MAX_GENERATIONS;
               generation++)
            {
                fprintf(best_fitness_file, "%f\n",
                   best_fitness_values[generation]);
            }
            fclose(best_fitness_file);
        }
        // Open a file to save the final population
        FILE *final_population_file = fopen("final_population.txt",
            "w");
        if (final_population_file)
        {
            for (int i = 0; i < POPULATION_SIZE; i++)
            {
                for (int j = 0; j < NUM_VARIABLES; j++)
                {
                    fprintf(final_population_file, "%f ",
                       population[i][j]);
                }
                fprintf(final_population_file, "\n");
            }
            fclose(final_population_file);
        }

        // Find the solution closest to (0, 0) using distance
           formual
        // Variables to hold the minimum distance and its index
        // Variables to hold the minimum values and their indices
        double min_x = fabs(final_population[0][0]);
        double min_y = fabs(final_population[0][1]);
        int min_index_x = 0;
        int min_index_y = 0;

        // Loop through the population to find the smallest
           absolute values
        for (int i = 0; i < POPULATION_SIZE; i++)
        {
            if (fabs(final_population[i][0]) < min_x)
            {
                min_x = fabs(final_population[i][0]);
                min_index_x = i;
            }
            if (fabs(final_population[i][1]) < min_y)
            {
                min_y = fabs(final_population[i][1]);
                min_index_y = i;
            }
        }

        printProgressBar(generation, MAX_GENERATIONS); // prints
           progress bar
    }
```

```
// Find the solution closest to (0, 0) using distance formula
// This should be after the final population has been
   completely assigned
double closest_solution_distance = __DBL_MAX__;

for (int i = 0; i < POPULATION_SIZE; i++)
{
    double x = population[i][0];
    double y = population[i][1];
    double distance = calculateDistance(x, y);

    if (distance < closest_solution_distance)
    {
        closest_solution_distance = distance;
        closest_solution_index = i;
    }
}



// <YOUR CODE: Jump to this part of code if the stopping
   criteria is met before MAX_GENERATIONS is met>

// ############################################################
   #######################
// You dont need to change anything here
// Here we print the CPU time taken for your code

printf("genetic algorithm complete\n");
printf("====================================\n");
printf("number of varibles: %d\n", NUM_VARIABLES);
for (int i = 0; i < 1; i++)
{
    printf("lower bound: %.10lf , %.10lf\n", Lbound[0], Lbound
       [1]);
}
```

```
    for (int i = 0; i < 1; i++)
    {
        printf("Upper bound: %.10lf ,%.10lf\n", Ubound[0], Ubound
           [1]);
    }

    printf("\n");

    printf("POPULATION_SIZE : %d\n", POPULATION_SIZE);
    printf("MAX GENERATIONS : %d\n", MAX_GENERATIONS);
    printf("crossover rate: %lf\n", crossover_rate);
    printf("mutate_rate : %lf\n", mutate_rate);
    printf("stop criteria : %lf\n", stop_criteria);

    end_time = clock();
    cpu_time_used = ((double)(end_time - start_time)) /
       CLOCKS_PER_SEC;
    printf("CPU time: %f seconds\n", cpu_time_used);
    // ##############################################################
       #######################

    // <Here print out the best solution and objective function
       value for the best solution like the format>

    printf("Best fitness value: %.10lf\n", best_fitness);
    if (closest_solution_index != -1)
    {
        printf("Best soltuon): (%.10lf, %.10lf)",
               population[closest_solution_index][0],
               population[closest_solution_index][1]);
    }

    printf("\n");

    return 0;
}

// chat gpt did this
```

```
void printProgressBar(int iteration, int total)
{
    int progressBarWidth = 50;
    float progress = (float)iteration / total;
    int numBarChars = (int)(progress * progressBarWidth);

    printf("Progress: [");
    for (int i = 0; i < numBarChars; i++)
    {
        putchar('=');
    }
    for (int i = numBarChars; i < progressBarWidth; i++)
    {
        putchar(' ');
    }
    printf("] %.1f%%\r", progress * 100);
    fflush(stdout);
}

double calculateDistance(double x, double y)
{
    return sqrt(x * x + y * y); // disrnace formula
}
double adaptiveMutationRate(int generation, int MAX_GENERATIONS,
   double startRate, double endRate)
{
    // Linearly decrease the mutation rate from startRate to
       endRate over the generations
    double rate = startRate + (endRate - startRate) * ((double)
       generation / (double)MAX_GENERATIONS);
    return rate < endRate ? endRate : rate; // Ensure that the rate
        never goes below the endRate
}

double adaptiveCrossoverRate(int generation, int MAX_GENERATIONS,
   double startRate, double endRate)
{
    // Linearly decrease the crossover rate from startRate to
       endRate over the generations
    double rate = startRate + (endRate - startRate) * ((double)
       generation / (double)MAX_GENERATIONS);
    return rate < endRate ? endRate : rate; // Ensure that the rate
        never goes below the endRate
}
```