NUMERICAL OPTIMISATION BY EVOLUTIONARY ALGORITHM

Submitted by:

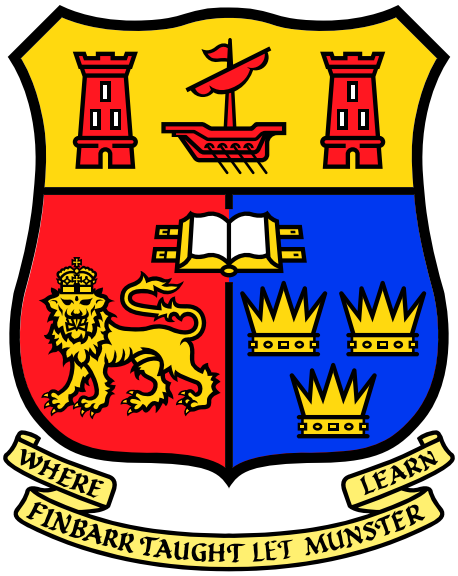
LUTONG ZHAO

Supervisor:

DR STEVEN PRESTWICH

Second Reader:

DR KEN BROWN



MSc Computing Science

School of Computer Science & Information Technology
University College, Cork

August 30, 2024

Lutong Zhao: *Numerical Optimisation by Evolutionary Algorithm,* MSc Computing Science, © August 30, 2024

Abstract

Many real life problems are related to mathematical functions, such as asset allocation and risk evaluation, and it is essential to optimize the mathematical functions to solve these problems.

This thesis focuses on optimizing a list of mathematical functions with evolutionary algorithms, and bringing up insightful thoughts and improvements based on the optimizations. Each mathematical function has its own characters: some functions do not have local minimum, such as Trid function, while some functions have many local minimum like Griewank function. The evolutionary algorithms involve genetic algorithm, differential evolution and particle swarm optimization. The three algorithms solve mathematical functions in different ways, and all of them are promising to optimize the functions.

In the experiments of optimizing mathematical functions, genetic algorithm, differential evolution and particle swarm optimization use the same number of evaluations, which guarantees the fairness. By tuning the parameters, we can learn how the parameters influence the optimization and find the best solutions to the mathematical functions. Besides the result calculated from the algorithms, the running time of the experiments is also important. Many factors may influence the running time, such as the complexity of the mathematical functions.

Based on the experiments of the mathematical functions, it is simple to find out each algorithm's performances on the distinct mathematical functions, which is helpful to have a deep comprehension in the algorithms and their attributes including the parameters. From the analysis of the experiment results, the advantages and disadvantages of the algorithms can be proved, and this may contribute to the future work.

Declaration

I confirm that, except where indicated through the proper use of citations and references, this is my original work and that I have not submitted it for any other course or degree.

Signed: _____

Lutong Zhao
August 30, 2024

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Numerical optimization by evolutionary algorithms is to solve objective functions with biological evolution and natural selection. This chapter will demonstrate the overview of the research area, motivation of the project and our final goal.

1.1 Thesis Overview

1.1.1 Overview of Research Area

The thesis's core idea is numerical optimization by Evolutionary Algorithm. In this project, the research area is solving the numerical optimization problem occurring in the real world, which could be across many different domains, such as finance and transportation.

In the finance industry, some tasks like asset allocation and risk evaluation are related to our research. For the area of transportation, optimizations are important as well, since we need to optimize the traffic flow and set better public transportation routes to provide more convenience.

The optimization means that we need to find the best solution for a given problem, and it can be solved by evolutionary algorithms, such as genetic algorithms.

1.1.2 Research Problem

The research problem is to compare different evolutionary algorithms or other heuristic algorithms by a list of multidimensional functions, and these multidimensional functions can be considered as real-world examples. The mathematical functions could be "Sum of Different Powers", "Zakharov Function", "Griewank Function".

The evolutionary algorithms can be Genetic Algorithms, Particle Swarm Optimization, and Differential Evolution, and they need to be optimized for the mathematical functions.

1.2 Thesis Motivation

By optimizing mathematical functions, we can evaluate different evolutionary algorithms' performance on each function, and summarize each algorithm's advantages and disadvantages, which helps us have a better understanding of the algorithms. Based on the characteristics of the evolutionary algorithms, we can find the best solutions with reasonable parameters and accurate implementations, and it helps us achieve optimization.

Another reason that we choose evolutionary algorithms as our primary solution to solve the mathematical functions is the robustness of this algorithm, which means that by using evolutionary algorithms, it is more likely to find the optimal solution in an uncertain environment, even if our initial parameters are not properly defined since it can learn from the previous calculations or implementations.

1.3 Thesis Mission

At the end of the thesis, the following aims need to be achieved.

The first aim is that for each mathematical function, we will find the best solutions with each algorithm, and try to minimize the running time utilization.

The second target is to compare different algorithms for solving the same mathematical functions and summarize each algorithm's advantages and disadvantages.

Another target is to think about how we can apply our algorithms to the real world, such as solving finance tasks like revenue prediction or finding the best traveling routes in the transportation industry.

1.4 Thesis Scope

The thesis focuses on optimizing mathematical functions with evolutionary algorithms, which are genetic algorithms, differential evolution and particle swarm optimization. The optimization refers to finding the global minimum values in this thesis.

As the evolutionary algorithms become time-consuming when optimizing large groups of data or large numbers of iterations, some parameters (i.e., population size, numbers of iterations) are defined.

Some mathematical functions may contain local optima, while others may not. For the former functions, it is important to avoid converging too early, and is necessary to balance the running time and the results.

1.5 Thesis Structure

The following four chapters demonstrates the thesis.

Chapter two focuses on background and basic concepts, which includes evolutionary algorithms, genetic algorithm, differential evolution and particle swarm optimization. This chapter also involves other concepts that are used in the thesis, such as mathematical functions.

Chapter three demonstrates the mathematical functions that will be optimized, including the challenges and trade-offs, and design approaches.

Chapter four displays the experiments results and analyzes each algorithm's performance on the optimized mathematical functions.

Chapter five analyzes the experiments, which includes the performances of genetic algorithm, differential evolution and particle swarm optimization in optimizing the mathematical functions. This chapter also compares the three evolutionary algorithms.

Chapter six is the conclusion part, which summarizes the experiment results and demonstrates several limitations found in the experiments. This chapter also brings up the future work.

Chapter 2

Background and Basic Concepts

2.1 Evolutionary Algorithm

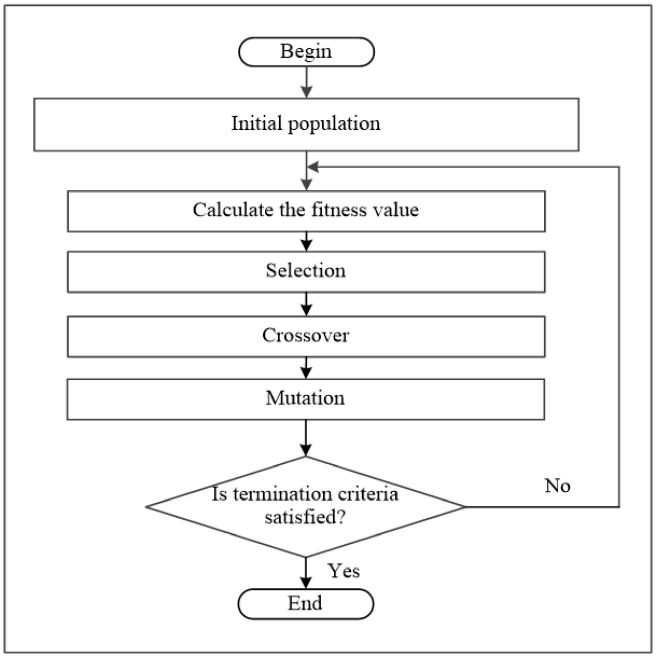
Evolutionary algorithms are intuited by the nature of selection and evolution and are widely used in optimization problems [Deb, Anand, and Joshi 2002]. For a general evolutionary algorithm, there are seven steps: initialization, selection, crossover, mutation, evaluation, death and survivor selection, and termination if conditions are met [Van Veldhuizen, and Lamont 1998]. In evolutionary algorithm, it holds a population of possible solutions, and after iterative training, an optimal or near-optimal solution will be generated. Evolutionary computation techniques get much attention for their ability to solve complex numerical functions [Charbonneau et al. 2002].

2.2 Genetic Algorithm

Genetic Algorithm is a type of Evolutionary algorithms, and used to solve optimization problems. The core ideas of genetic algorithms are natural selection and biological evolution. From figure 2.1, we can find there are five steps for a typical genetic algorithm: population initialization, scoring population, retaining elite, parents selection, producing crossover and mutation children [Lambora, Gupta, and Chopra 2019]. After each iteration, mutated children may have better performance than their parents, and getting closer to the optimal solution. With properly implementing genetic algorithm, it is promising to get the global optimal solution. Moreover, genetic algorithm supports parallelism, which can improve the efficiency of the optimization process [Katoch, Chauhan, and Kumar 2021].

In genetic algorithm, selection method is an important function to determine whether an individual is to be selected for the next generation. There are several traditional selections methods, roulette wheel selection, rank selection, steady state selection and tournament selection [Chudasama, Shah, and Panchal 2011]. In the experiments, we are using tournament selection as the selection method. Tournament selection is to select the fittest individual of the set for the next generation [Jebari, Madiafi, et al. 2013].

Figure 2.1: Flow chart of Genetic Algorithms [Albadr et al. 2020]



2.3 Differential Evolution

Differential evolution (DE) is another type of Evolutionary algorithm, and is efficient to solve optimization problems. Differential evolution has simple structure, and its main operations for differential evolution are selection, mutation, and crossover [Gämperle, Müller, and Koumoutsakos 2002]. Compared with genetic algorithms, differential evolution uses a different strategy in mutation. Differential evolution generates the mutation children by comparing differences of their the parents, while genetic algorithms determines the probability when selecting children. Therefore, differential evolution is more influenced by the approximation effect than genetic algorithms[Yuan et al. 2023].

Differential evolution has the characteristic of exploring the whole searching space, which guarantees that it can find the global optima. Differential evolution takes advantage of the population for iterations and avoids converging to local optima, and this attribute makes it more robust in terms of global optimization [Storn 1996]. DE also has the scalability to the dynamic dimensions so that it can optimize high-dimensionality problem efficiently [Rocca, Oliveri, and Massa 2011]. Differential evolution is simple to adjust parameters and performs well in optimizing mathematical functions based on its attributes.

2.4 Particle Swarm Optimization

Particle swarm optimization (PSO) is a population-based stochastic optimization algorithm, and researchers first discovered this algorithm in 1995 [Juneja, and Nagar 2016]. The researchers observed group behaviors of some kinds of animals, such as flocks of birds or schools of fish, and based on these actions, they brought up this optimization algorithm [Wang, Tan, and Liu 2017].

Particle swarm optimization has five steps. The first step is to initialize a group of particles, constants and parameters. Each particle can be considered as a potential solution. The second step is to update velocity for each particle by finding their optimal position. The third step is to update position for each particle, regarding to their velocity. The fourth step is evaluation, which needs to use fitness functions to compare each particle. When the optimal result is found or meeting the stopping criterion, the algorithm will be terminated [Eberhart, and Kennedy 1995].

Particle swarm optimization has the ability to optimize the numerical functions based on its characteristics. Particle swarm optimization can solve high-dimensional problems, as it finds global optima efficiently when the searching space becomes large compared with other algorithms. PSO can avoid the curse of dimensionality [Sedighizadeh, and Masehian 2009]. Griewank function has the attribute of high dimensionality, and particle swarm optimization has good performance on optimizing it. Another reason is that PSO has stable and consistent performance on finding global optima in mathematical function, as it might not be struggled by the local optima. Particle swam optimization is not complex to implement and has high adaptability.

2.5 Mathematical Benchmark Functions

Mathematical Benchmark Functions are used to evaluate the performance of optimization algorithms and help us to understand each algorithm's characteristics. Some typical mathematical benchmark functions are the Rosenbrock Function, Sphere Function and Ackley Function.

2.6 Fitness Function

In numerical optimization, in order to evaluate the performance of a solution, a fitness function needs to be designed. The fitness function helps evolutionary algorithms select more promising solutions, and plays an important role in iterative evolution [Baresel, Sthamer, and Schmidt 2002]. In genetic algorithms, fitness function is used to score the generated children, and the children that with higher scores are more likely to keep alive for the later generations.

Chapter 3

Literature Review

This chapter demonstrates the mathematical functions that will be optimized, including the challenges and trade-offs, and design approaches.

3.1 Problem Context and Methodological Framework

3.1.1 Problem Description

In this project, the numerical optimization problems refer to mathematical functions, which need to be optimized for either maximum or minimum value by evolutionary algorithms. There are three functions that will be optimized, Sum of Different Powers, Zakharov Function, and Griewank Function, which will be introduced in the following sections.

3.1.2 Sum of Different Powers Function

The function is to calculate the total of powers for a given list of numbers, whose domain is usually in [-1, 1]. The mathematical formula of the "Sum of Different Powers Function" is $f(x) = \sum_{i=1}^{n} |x_i|^{i+1}$. The function is displayed by figure 3.1.

3.1.3 Zakharov Function

Zakharov function does not have local minimum value, but there exists global minimum. The function can be expressed as $f(\mathbf{x}) = \sum_{i=1}^{n} x_i^2 + (\sum_{i=1}^{n} 0.5 \times i \times x_i)^2 + (\sum_{i=1}^{n} 0.5 \times i \times x_i)^4$. The function is displayed by figure 3.2.

3.1.4 Griewank Function

Griewank function is a typical benchmark function for optimization algorithms, which contains one global minima and many local minima. This function is not linear as it contains cosine function.

Figure 3.1: Sum of Different Powers Function - 3D Plot

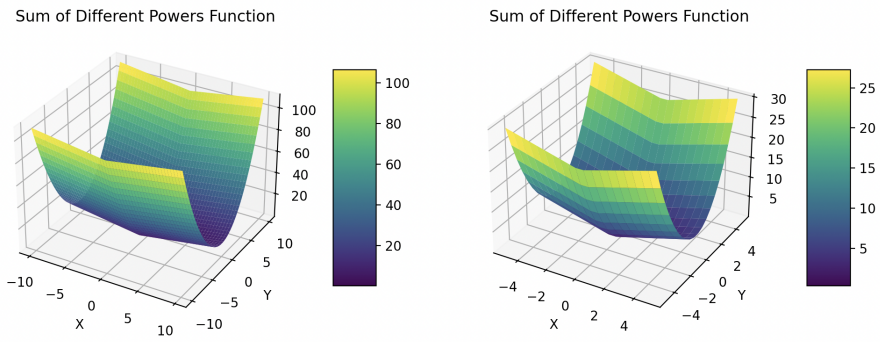


Figure 3.2: Zakharov Function - 3D Plot

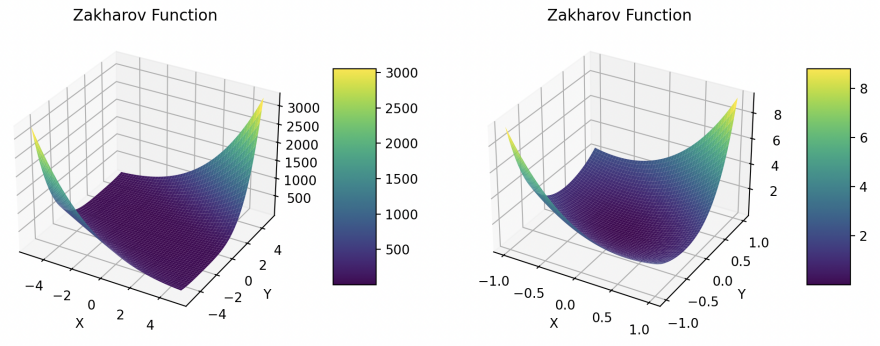
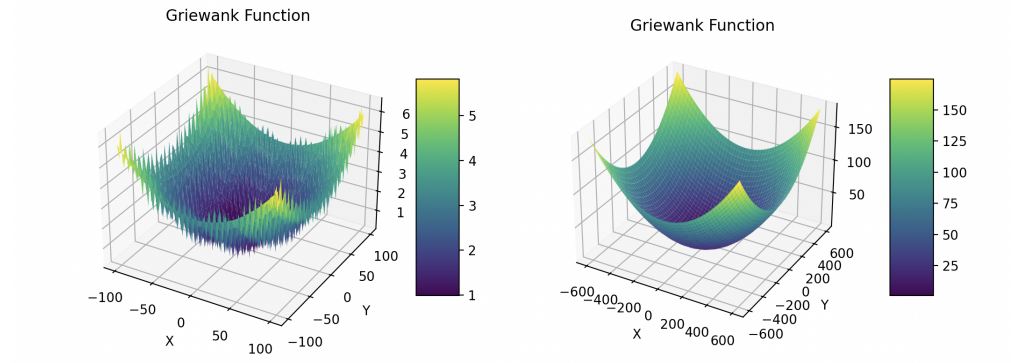


Figure 3.3: Griewank Function - 3D Plot



Griewank function can be written as $f(\mathbf{x}) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$. The function is displayed by figure 3.3.

3.1.5 Schwefel Function

Schwefel function contains many local minima and one global minima.

Schwefel function can be written as

$$f(\mathbf{x}) = 418.9829 \times n - \sum_{i=1}^{n} x_i \sin(\sqrt{|x_i|})$$

. The function is displayed by figure 3.4.

3.1.6 Rastrigin Function

Rastrigin function contains many local minima and one global minima.

Rastrigin function can be written as

$$f(\mathbf{x}) = 10 \times n + \sum_{i=1}^{n} (x_i^2 - 10 \times \cos(2 \times \pi \times x_i))$$

. The function is displayed by figure 3.5.

3.1.7 Sphere Function

Sphere function has one global minima and no local minima, which is displayed by figure 3.6.

Figure 3.4: Schwefel Function - 3D Plot

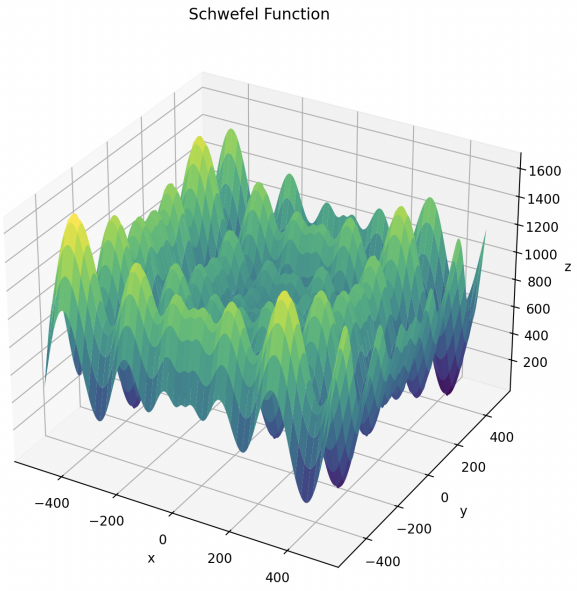


Figure 3.5: Rastrigin Function - 3D Plot

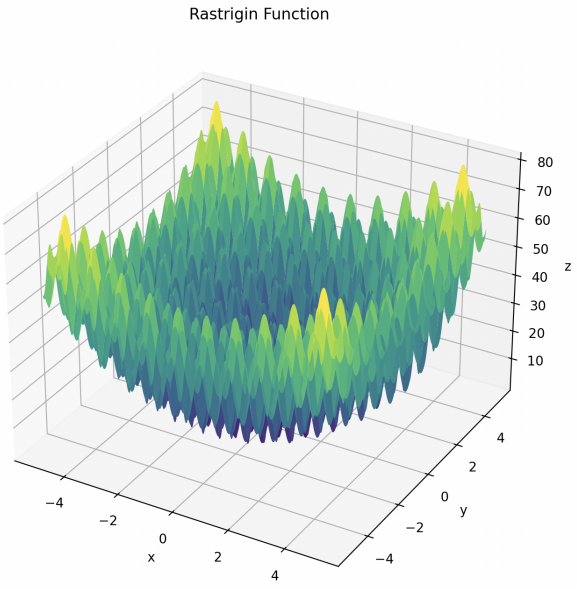
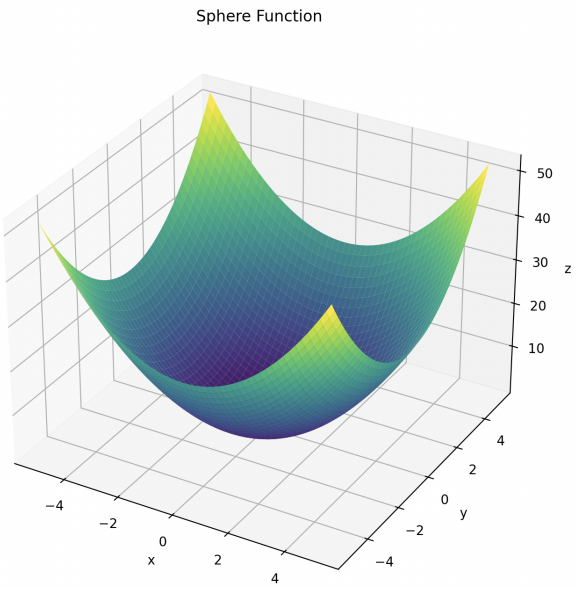


Figure 3.6: Sphere Function - 3D Plot



Sphere function can be written as

$$f(\mathbf{x}) = \sum_{i=1}^{n} x_i^2$$

.

3.1.8 Sum Squares Function

Sum squares function has one global minima and no local minima, which is displayed by figure 3.7.

Sum squares function can be written as

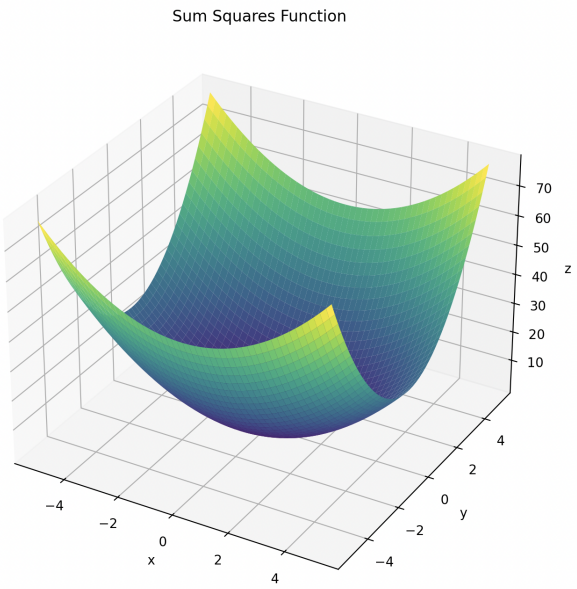
$$f(\mathbf{x}) = \sum_{i=1}^{n} i \times x_i^2$$

.

3.1.9 Trid Function

Trid function has one global minima and no local minima, which is displayed by figure 3.8.

Figure 3.7: Sum Squares Function - 3D Plot



Trid function can be written as

$$f(\mathbf{x}) = \sum_{i=1}^{n} (x_i - 1)^2 - \sum_{i=2}^{n} x_i \times x_{i-1}$$

.

3.1.10 Booth Function

Booth function has one global minima and no local minima, which is displayed by figure 3.9.

Booth function can be written as

$$f(\mathbf{x}) = (x_1 + 2 \times x_2 - 7)^2 + (2 \times x_1 + x_2 - 5)^2$$

.

3.1.11 Michalewicz Function

Michalewicz function has one global minima and many local minima, which is displayed by figure 3.11.

Figure 3.8: Trid Function - 3D Plot

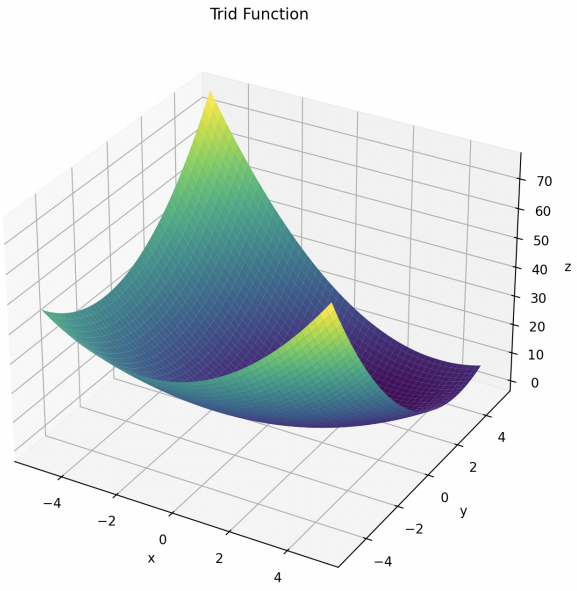


Figure 3.9: Booth Function - 3D Plot

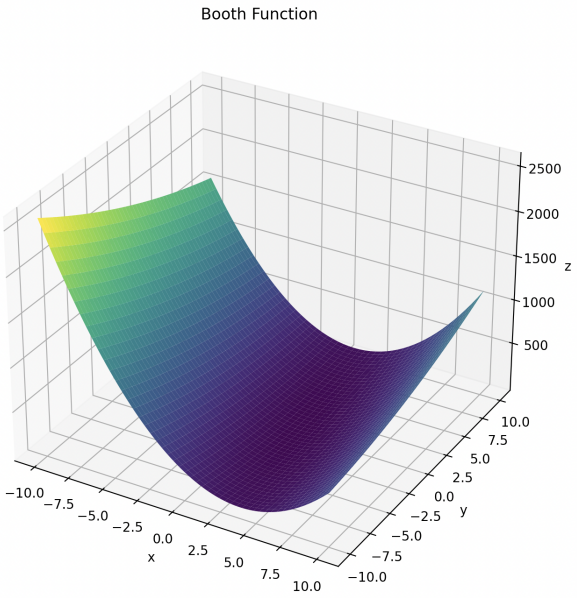
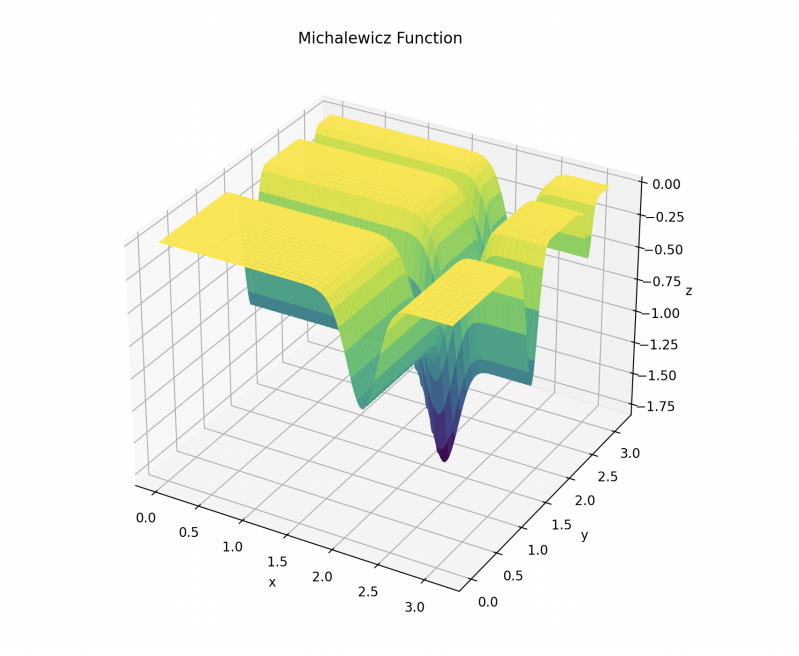


Figure 3.10: Michalewicz Function - 3D Plot



Michalewicz function can be written as

$$f(\mathbf{x}) = -\sum_{i=1}^{n} \sin(x_i) \times \sin(\frac{i \times x_i^2}{\pi})^{2m}$$

.

3.1.12 Matyas Function

Matyas function has one global minima and no local minima, which is displayed by figure 3.11.

Matyas function can be written as

$$f(\mathbf{x}) = 0.26 \times (x_1^2 + x_2^2) - 0.48 \times x_1 \times x_2$$

.

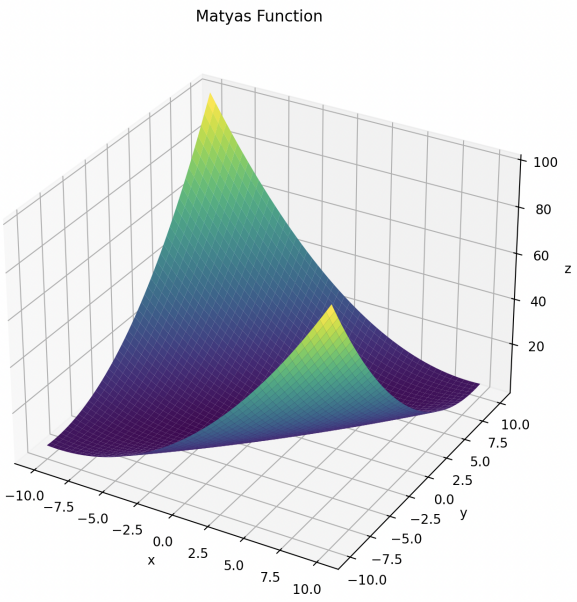
3.1.13 McCormick Function

McCormick function has one global minima and no local minima, which is displayed by figure 3.12.

McCormick function can be written as

$$f(\mathbf{x}) = \sin(x_1 + x_2) + (x_1 - x_2)^2 - 1.5 \times x_1 + 2.5 \times x_2 + 1$$

Figure 3.11: Matyas Function - 3D Plot



.

3.1.14 Cross in Tray Function

Cross in Tray function has multiple global minima and many local minima, which is displayed by figure 3.13.

Cross in Tray function can be written as

$$f(\mathbf{x}) = -0.0001 \times (|\sin x_1 \times \sin x_2| \times \exp|100 - \frac{\sqrt{x_1^2 + x_2^2}}{\pi}| + 1)^{0.1}$$

.

3.1.15 Drop Wave Function - 3D Plot

Drop Wave function has one global minima and many local minima, which is displayed by figure 3.13.

Drop Wave function can be written as

$$f(\mathbf{x}) = -\frac{1 + \cos(12 \times \sqrt{x_1^2 + x_2^2})}{0.5 \times (x_1^2 + x_2^2) + 2}$$

.

Figure 3.12: McCormick Function - 3D Plot

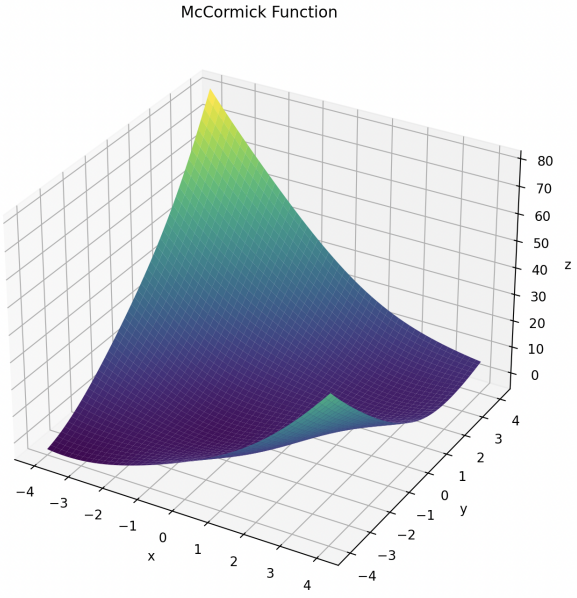


Figure 3.13: Cross in Tray Function - 3D Plot

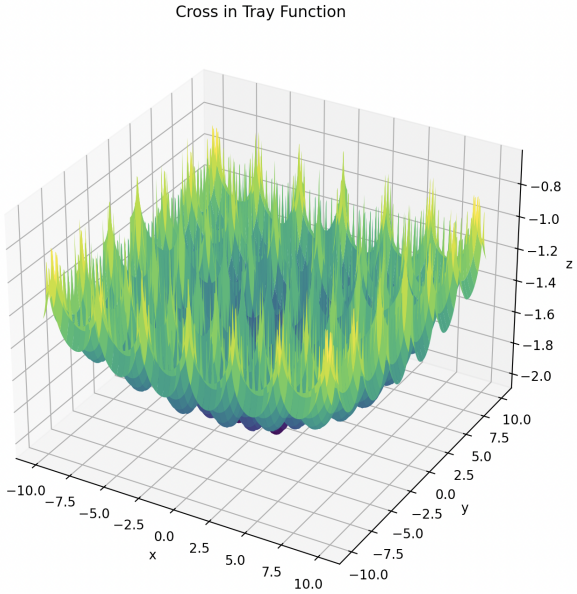
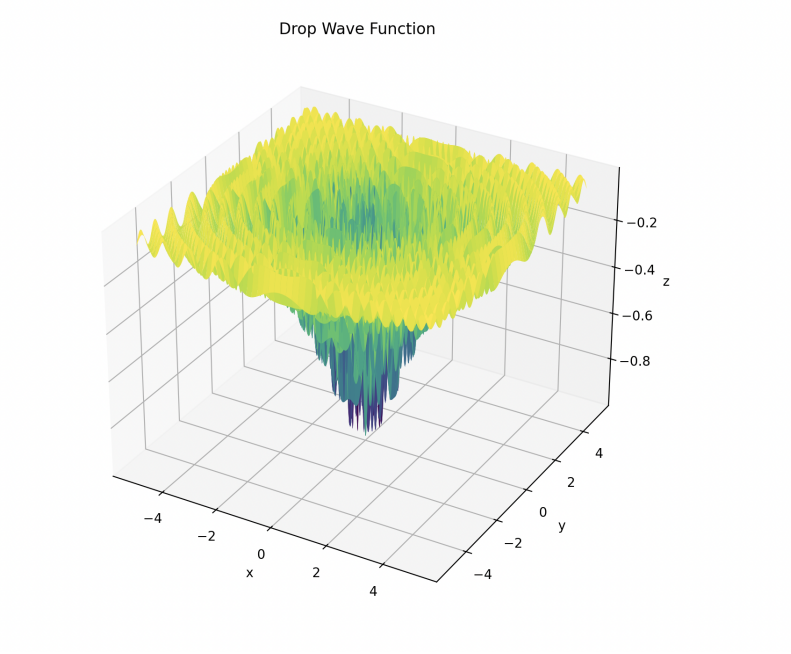


Figure 3.14: Drop Wave Function - 3D Plot



-abs(np.sin(x1) * np.cos(x2) * np.exp(abs(1 - (x1 ** 2 + x2 ** 2) ** 0.5 / np.pi)))

3.1.16 Holder Table Function - 3D Plot

Holder table function has 4 global minima and many local minima, which is displayed by figure 3.15.

Holder table function can be written as

$$-|\sin x_1 \times \cos x_2 \times \exp(|1 - \frac{\sqrt{x1^2 + x2^2}}{\pi}|)|$$

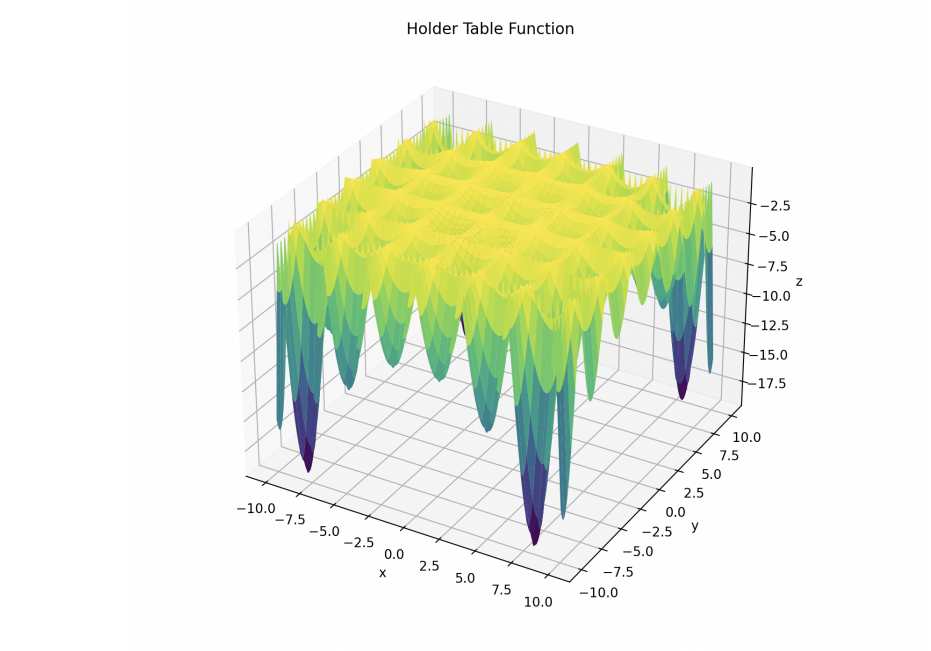
.

3.1.17 Challenges and Trade-offs

There are a few challenges when we optimize the mathematical functions. One is choosing proper algorithms. There are plenty of evolutionary algorithms and other similar algorithms available to use, and we need to find two or three that have good performance. Another challenge is how to improve the algorithms or find the optimal solution, and one possible solution is to tune the parameters with large numbers of training.

One trade-off in the optimization is between space complexity and time complexity. We may declare more variables to decrease the time complexity of the algorithm. An-

Figure 3.15: Holder Table Function - 3D Plot



other trade-off is the running time and the accuracy. The more time we run and tune parameters, the better performance we may get.

3.1.18 Summary of Solution Approaches

To solve the mathematical functions, evolutionary algorithms can be applied. By implementing genetic algorithms, differential evolution algorithm and other evolutionary algorithms, we may find the optimal solution for the given problems. Another possible way is to find other potential algorithms or develop a new algorithm based on the thesis of genetic algorithms or other algorithms.

3.1.19 Datasets

In this project, the mathematical functions are existing benchmark functions, and they are well-defined. The functions' optima are provided to researchers for training algorithms, and we can take this advantage to get to know each characteristic of each evolutionary algorithm we will develop.

3.1.20 Performance Metrics

For developing evolutionary algorithms, performance metrics are used to do the evaluation part. One performance metric that will be used is efficiency of the algorithms, which

records the running time for each testing, and it is useful when comparing different algorithms on solving the same problem. We will also evaluate how fast each algorithm converge to their final solution, as it suggests how well an algorithm performs in each iteration.

3.2 Design Approaches

From the perspective of fitness function design for numerical optimization, there are two potential design approaches. One is the constraint handling, and the other one is the objective aggregation. The two approaches have different methodologies in optimizing numerical problems.

3.2.1 Constraint Handling

The idea of constraint handling in the fitness function is to penalize the solutions that do not satisfy the constraints rules [Deb 2000]. In the implementation of the evolutionary algorithm, it is necessary to minimize the fitness function with penalization to get the global optimal solution.

Constraint handling has a few advantages. One advantage is that it can guarantee that valid solutions will be obtained when we use evolutionary algorithms to optimize the mathematical functions as constraint handling follows the problems' requirements. Another advantage is that it may help the evolutionary algorithms find the global optimal solution as it improves the convergence. One disadvantage of constraint handling is that if it is not defined well or implemented properly in optimizing the numerical problems, it may cause bad performance. Another disadvantage is that using constraint handling for the fitness function will increase the complexity of the algorithms. It is necessary to consider the trade-offs between the efficiency and complexity of solving the mathematical functions.

3.2.2 Objective Aggregation

The objective aggregation methodology gathers different objective functions and generates one fitness value, and each function has its weight [Jin, Olhofer, and Sendhoff 2001]. The objective aggregation reduces the complexity of the numerical optimization and considers more objective functions compared to the single objective function.

One advantage of objective aggregation is that it can be implemented straightforwardly by following its principles. Another advantage is that it may optimize numerical problems more effectively and efficiently as it combines several objectives into one. One disadvantage of objective aggregation is that in the process of the aggregation, some information is lost, which might cause the evolutionary algorithm to lack some key dimensions of the numerical problems. Another disadvantage is that objective aggregation does not guarantee the global optimal solution will be obtained.

3.3 Comparative Analysis

Our research focuses on solving mathematical functions by evolutionary algorithms, while the previous papers pay more attention to optimizing numerical problems. These aims are similar as all of them use heuristic algorithms to optimize problems, and the subtle difference is that our research can be applied to scenarios in the real world by finding the optimal value of a mathematical function, which can reveal some implications in large sets of numbers.

Chapter 4

Implementation and Results

15 mathematical functions will be optimized by 3 evolutionary algorithms. The evolutionary algorithms are genetic algorithm, differential evolution and particle swarm optimization.

4.1 Environment Setup

The programming language is Python3. All of the experiments were conducted on the Colab. The capacity of RAM is 12.67G.

To keep fairness of comparing algorithms on the same mathematical functions, the numbers of evaluations for each algorithm are the same. By default, the number of evaluation is 10000, and in optimizing some functions, the number of evaluation is changed for the three algorithms.

4.1.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used with version 1.4.1. DEAP is a computational framework for evolutionary algorithms, which is useful in optimizing mathematical functions[*DEAP 1.4.1 documentation* 2023].

The implemented genetic algorithm is the generational genetic algorithm. The number of evaluations is equal to population size times the number of generations.

When using the genetic algorithms, the following 2 parameters are set to default values. The reason is that the larger the values will get the better results, and there is no advantage to tune these 2 parameters.

- **Population Size**: the number of solutions in the population. In the experiments, the value is set to 100.
- **Number of Generations**: the maximum number of iterations for the optimization. The number of generations is set to 100.

The following are the parameters that need to be optimized in genetic algorithms:

- **Probability of Crossover**: the probability of crossing two solutions.
- **Probability of Mutation**: the probability of changing a solution.

In the experiment of genetic algorithms optimizing mathematical functions, four testing values for the each above parameters are used, therefore, there are 64 (8 * 8) combinations. For instance: (crossover rate: 0.6, mutation rate: 0.05). Following are the values involved in the experiment:

- **Probability of Crossover**: 0.05, 0.1, 0.2, 0.3, 0.5, 0.6, 0.8, 1.0
- **Probability of Mutation**: 0.01, 0.05, 0.1, 0.25, 0.4, 0.6, 0.75, 0.9

4.1.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize library with version 1.14.0.

For the differential evolution, the number of evaluations is equal to population size times the number of iterations.

When using the differential evolution optimizing mathematical functions, the 2 parameters are set to default values. The reason is that the larger the values will get the better results, and it is reasonable to keep them fixed.

- **Population Size**: the number of solutions in the population. In the experiments, the value is set to 100.
- **Number of iterations**: the maximum number of iterations for the optimization. The number of iterations is set to 100.

The following are the parameters that need to be optimized in differential evolution:

- **Tolerance**: tolerance of convergence
- **Recombination**: the probability of recombination, within the range [0, 1]
- **Mutation**: accept a float or a tuple containing lower bound and upper bound, which decides the search radius: when increasing the value of mutation, the search radius becomes larger as well, which will take more time to converge.

In the experiment of differential evolution optimizing mathematical functions, three testing values for the each above parameters are used, therefore, there are 64 (4 * 4 * 4) combinations. For instance: (tolerance: 0.01, recombination: 0.6, mutation: (0.5, 1.0)). Following are the values involved in the experiment:

- **Tolerance**: 0.01, 0.02, 0.05, 0.1
- **Recombination**: 0.5, 0.6, 0.8, 1.0
- **mutation**: (0.5, 1), (0.75, 1), (0.5, 1.5), (0.75, 1.5)

4.1.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms with the version 1.3.0.

For the particle swarm optimization, the number of evaluations is equal to number of particles times the number of iterations.

The following are the parameters that need to be optimized in particle swarm optimization:

- **Options**: combining three parameters, cognitive parameter, social parameter and inertia parameter.
- **Number of Particles**: number of particles used in the swarm

In the experiment of particle swarm optimization solving mathematical functions, five testing values for the each above parameters are used, therefore, there are 36 (6 * 6) combinations. For instance: (options: ['c1': 1.0, 'c2': 1.0, 'w': 0.15], number of particles: 50). Following are the values involved in the experiment:

- **Options**: ['c1': 1.0, 'c2': 1.0, 'w': 0.15]

  ['c1': 1.0, 'c2': 0.5, 'w': 0.15]

  ['c1': 0.5, 'c2': 1.0, 'w': 0.15]

  ['c1': 0.5, 'c2': 1.0, 'w': 0.25]

  ['c1': 0.5, 'c2': 0.5, 'w': 0.15]

  ['c1': 1.0, 'c2': 1.0, 'w': 0.25]
- **Number of Particles**: 10, 30, 50, 70, 100, 200

4.2 Sum of Different Powers

In experiments, a list of numbers will be used. The maximum of the number is 5, and the minimum number is -5.

4.2.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used.

The table 4.1 are the top 10 optimization solutions in the 64 solutions.

From the table 4.1, the following conclusion can be made:

- The best solution is (crossover rate: 0.6, mutation rate: 0.25), and its result is 7.531666490795148e-06.
- This experiment took 18 seconds to optimize the mathematical functions.

Table 4.1: Genetic Algorithm on Sum of Different Powers

Crossover Rate	Mutation Rate	Result
0.6	0.25	7.531666490795148e-06
0.3	0.75	9.651054138592353e-06
0.2	0.9	1.1149284203778376e-05
0.05	0.4	2.1854168235743568e-05
0.2	0.75	2.5698788149685e-05
0.5	0.6	3.113236933210791e-05
0.2	0.6	3.263799967919651e-05
1.0	0.75	3.305754146998692e-05
0.3	0.9	3.750097993053178e-05
0.6	0.6	3.8228833438561255e-05

- In terms of crossover rate, 0.6 is the best value compared with other 7 values (i.e., 0.05, 0.1, 0.2, 0.3, 0.5, 0.8, 1.0), as it appears 2 times in the top ten, and also in the best solutions.
- For the mutation rate, 0.25 has good performance as it is used by the best solution. The value 0.75 also performs well in optimizing the sum of different powers function as it appears 3 times in the top 10 solutions.

4.2.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize.

The table 4.2 are the top 10 optimization solutions in the 64 solutions.

Table 4.2: Differential Evolution on Sum of Different Powers

Tolerance	Recombination	Mutation	result
0.01	0.5	(0.5, 1)	0.0
0.01	0.5	(0.5, 1.5)	0.0
0.01	0.5	(0.75, 1)	0.0
0.01	0.5	(0.75, 1.5)	0.0
0.01	0.6	(0.5, 1)	0.0
0.01	0.6	(0.5, 1.5)	0.0
0.01	0.6	(0.75, 1)	0.0
0.01	0.6	(0.75, 1.5)	0.0
0.01	0.8	(0.5, 1)	0.0
0.01	0.8	(0.5, 1.5)	0.0

From the table 4.2, the following conclusion can be made:

- Differential evolution finds the global minimum value 0.0 .
- This experiment took 54 seconds to optimize the mathematical functions.
- Differential evolution performs well in optimizing sum of different powers.

4.2.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms.
The table 4.3 are the top 10 optimization solutions in the 36 solutions.

Table 4.3: Particle Swarm Optimization on Sum of Different Powers

Options	Particles	result
'c1': 1.0, 'c2': 0.5, 'w': 0.15	25	0.04462981669088782
'c1': 1.0, 'c2': 1.0, 'w': 0.15	10	0.625085677820622
'c1': 1.0, 'c2': 1.0, 'w': 0.15	50	0.7675521364370064
'c1': 1.0, 'c2': 0.5, 'w': 0.15	10	4.228411582857754
'c1': 1.0, 'c2': 1.0, 'w': 0.25	200	7.8773635148091525
'c1': 0.5, 'c2': 0.5, 'w': 0.15	25	8.13546701891827
'c1': 0.5, 'c2': 0.5, 'w': 0.15	50	97.18562402222555
'c1': 0.5, 'c2': 0.5, 'w': 0.15	10	948.864639463704
'c1': 0.5, 'c2': 1.0, 'w': 0.15	10	3462.08772346766051
'c1': 0.5, 'c2': 1.0, 'w': 0.25	10	15054.731104634317

From the table 4.3, the following conclusion can be made:

- The best solution is (options: ['c1': 1.0, 'c2': 0.5, 'w': 0.15], number of particles: 25), and result is 0.04462981669088782
- This experiment took 47 seconds to optimize the mathematical functions.
- The best options is ['c1': 1.0, 'c2': 0.5, 'w': 0.15], as it appears 2 times in the table, and it is used in the best solution.
- In terms of number of particles, the best value is 25, as it is used by the best solution, and it much smaller than the other solutions.

4.2.4 Comparative Analysis

Based on the above three algorithms optimizing the Sum of Different Powers function, following conclusions can be made:

- Differential evolution has the best performance compared with the other two algorithms, as its best solution's result is 0.0, while genetic algorithm get the minimum cost 7.531666490795148e-06 and particle swarm optimization's minimum cost is 0.04462981669088782.
- From the perspective of the difference between the best solution to the 10th solution, the particle swarm optimization has the biggest differences, which suggests that its solutions have more diversity. The minimum cost for the differential evolution is 0.04462981669088782, and the cost of the 10th solution is 15054.731104634317, which is much larger than the cost of best solution as the difference of the magnitude is 7. For the genetic algorithm, the difference of the magnitudes between the best solution and the 10th solution is 1, and 0 for the differential evolution.
- All of the three algorithms takes less than 1 minute. Differential evolution takes more time (54s) than genetic algorithm (18s) and particle swarm optimization (47s). One reason is that differential evolution runs more experiments (64) compared with that of particle swarm optimization (36), and the same as that of genetic algorithm (64). Another reason is that the convergence speed of differential evolution is slower than the other two algorithms.

4.3 Zakharov Function

In experiments, a list of floats will be used. The maximum of the number is 5, and the minimum number is -5.

4.3.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used.

The table 4.4 are the top 10 optimization solutions in the 64 solutions.

From the table 4.4, the following conclusion can be made:

- The best solution is (crossover rate: 1.0, mutation rate: 0.01), and its result is 0.03603144484075108.
- This experiment took 19 seconds to optimize the mathematical functions.
- In terms of crossover rate, the value 0.6 is used by the best solution. The value 0.5 also performs well in optimizing Zakharov function as it appears twice in the top 4 solutions.
- For the mutation rate, the best value is 0.9, as it appears 4 times in the top 10 solutions and used by the best solution. The value 0.75 also performs well in optimizing Zakharov function as it appears 4 times in the table.

Table 4.4: Genetic Algorithm on Zakharov Function

Crossover Rate	Mutation Rate	Result
0.6	0.9	0.03603144484075108
0.5	0.75	0.163851735277469
0.3	0.75	0.21919115428349392
0.5	0.4	0.25913244685118625
0.1	0.9	0.29774981204823553
0.5	0.9	0.39662696452321516
0.3	0.6	0.41578645689376764
1.0	0.9	0.45837103970388543
1.0	0.75	0.4691909141265021
0.2	0.75	0.5295734998087809

4.3.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize library.

The table 4.5 are the top 10 optimization solutions in the 64 solutions.

Table 4.5: Differential Evolution on Zakharov Function

Tolerance	Recombination	Mutation	result
0.01	0.5	(0.5, 1)	0.0
0.01	0.5	(0.5, 1.5)	0.0
0.01	0.5	(0.75, 1)	0.0
0.01	0.5	(0.75, 1.5)	0.0
0.01	0.6	(0.5, 1)	0.0
0.01	0.6	(0.5, 1.5)	0.0
0.01	0.6	(0.75, 1)	0.0
0.01	0.6	(0.75, 1.5)	0.0
0.01	0.8	(0.5, 1)	0.0
0.01	0.8	(0.5, 1.5)	0.0

From the table 4.5, the following conclusion can be made:

- Differential evolution finds the global minimum value 0.0 .
- This experiment took 52 seconds to optimize the mathematical functions.
- Differential evolution performs well in optimizing Zakharov powers.

4.3.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms.
The table 4.6 are the top 10 optimization solutions in the 36 solutions.

Table 4.6: Particle Swarm Optimization on Zakharov Function

Options	Particles	result
'c1': 1.0, 'c2': 1.0, 'w': 0.25	10	49.18337615593967
'c1': 1.0, 'c2': 1.0, 'w': 0.15	25	81.79816824470767
'c1': 1.0, 'c2': 1.0, 'w': 0.15	10	131.34918126740297
'c1': 0.5, 'c2': 0.5, 'w': 0.15	25	141.38324019838439
'c1': 0.5, 'c2': 0.5, 'w': 0.15	50	1381.0683422753355
'c1': 0.5, 'c2': 1.0, 'w': 0.25	10	2192.0865722615517
'c1': 1.0, 'c2': 0.5, 'w': 0.15	10	8641.364213360306
'c1': 0.5, 'c2': 1.0, 'w': 0.15	10	14542.630251711167
'c1': 0.5, 'c2': 1.0, 'w': 0.15	25	216232.46938013472
'c1': 0.5, 'c2': 0.5, 'w': 0.15	10	264025.1846559072

From the table 4.6, the following conclusion can be made:

- The best solution is (options: ['c1': 1.0, 'c2': 1.0, 'w': 0.25], number of particles: 10), and result is 49.18337615593967
- This experiment took 47 seconds to optimize the mathematical functions.
- The best options are ['c1': 1.0, 'c2': 1.0, 'w': 0.25], as it is used in the best solution. The options ['c1': 1.0, 'c2': 1.0, 'w': 0.15] also performs well since it appears 3 times in the top 10 solutions.
- In terms of number of particles, the best value is 10, as it appears 6 times in the top 10 solutions. The function may not need many particles, and small number of particles can get good optimization.

4.3.4 Comparative Analysis

Based on the above three algorithms optimizing the Zakharov function, following conclusions can be made:

- Differential evolution has the best performance compared with the other two algorithms, as its best solution's result is 0.0, while genetic algorithm gets the minimum cost 0.03603144484075108 and particle swarm optimization's minimum cost is 49.18337615593967.

- From the perspective of the difference between the best solution to the 10th solution, the particle swarm optimization has the biggest differences, which suggests that its solutions have more diversity. The minimum cost for the particle swarm optimization is 49.18337615593967, and the cost of the 10th solution is 264025.1846559072, which is much larger than the cost of best solution as it is over 5000 times larger than the lowest cost. While for the differential evolution and genetic algorithm, the differences between the cost of the 10th solution and the lower cost are no more than 17 times.
- All of the three algorithms takes less than 1 minute. Differential evolution takes more time (52s) than genetic algorithm (19s) and particle swarm optimization (47s). One reason is that differential evolution runs more experiments (64) compared with that of particle swarm optimization (36), and the same as that of genetic algorithm (64). Another reason is that the convergence speed of differential evolution is slower than the other two algorithms.

4.4 Griewank Function

In experiments, a list of floats will be used. The maximum of the number is 5, and the minimum number is -5.

4.4.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used.

The following table are the top 10 optimization solutions in the 64 solutions.

Table 4.7: Genetic Algorithm on Griewank Function

Crossover Rate	Mutation Rate	Result
1.0	0.01	4.353184479555239e-13
0.8	0.05	9.212630658339549e-13
1.0	0.1	1.4286460903178977e-11
1.0	0.05	1.7306378552461865e-11
0.6	0.25	1.8170324123101267e-07
0.8	0.4	9.621398637804646e-07
1.0	0.4	9.687155776427758e-07
1.0	0.25	1.40751739174938e-06
0.6	0.01	5.7008901739319384e-05
0.3	0.4	0.00025694412231591635

From the table 4.7, the following conclusion can be made:

- The best solution is (crossover rate: 1.0, mutation rate: 0.01), and its result is 4.353184479555239e-13.
- This experiment took 41 seconds to optimize the mathematical functions.
- In terms of crossover rate, the value 1.0 is used by the best solution, and it appears 5 times in the top 10 solutions.
- For the mutation rate, the best value is 0.01, as it appears 2 times in the top 10 solutions and used by the best solution. The value 0.05 also performs well in optimizing Griewank function as it appears 2 times in the table. Small mutation rates have good performance in optimizing this mathematical function.

4.4.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize library.

The following table are the top 10 optimization solutions in the 64 solutions.

Table 4.8: Differential Evolution on Griewank Function

Tolerance	Recombination	Mutation	result
0.01	0.5	(0.5, 1)	0.0
0.01	0.5	(0.5, 1.5)	0.0
0.01	0.5	(0.75, 1)	0.0
0.01	0.5	(0.75, 1.5)	0.0
0.01	0.6	(0.5, 1)	0.0
0.01	0.6	(0.5, 1.5)	0.0
0.01	0.6	(0.75, 1)	0.0
0.01	0.6	(0.75, 1.5)	0.0
0.01	0.8	(0.5, 1)	0.0
0.01	0.8	(0.5, 1.5)	0.0

From the table 4.8, the following conclusion can be made:

- Differential evolution finds the global minimum value 0.0 .
- This experiment took 28 seconds to optimize the mathematical functions.
- Differential evolution performs well in optimizing Griewank powers.

4.4.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms.

The table 4.9 are the top 10 optimization solutions in the 36 solutions.

From the table 4.9, the following conclusion can be made:

Table 4.9: Particle Swarm Optimization on Griewank Function

Options	Particles	result
'c1': 1.0, 'c2': 0.5, 'w': 0.15	100	0.0020883614348142787
'c1': 0.5, 'c2': 1.0, 'w': 0.15	10	0.04959724668060772
'c1': 1.0, 'c2': 0.5, 'w': 0.15	200	0.065285335662153
'c1': 0.5, 'c2': 1.0, 'w': 0.15	80	0.08791037547013525
'c1': 1.0, 'c2': 1.0, 'w': 0.25	100	0.12020580512392842
'c1': 1.0, 'c2': 1.0, 'w': 0.25	50	0.5078335754729641
'c1': 0.5, 'c2': 0.5, 'w': 0.15	10	0.5543524387727676
'c1': 0.5, 'c2': 1.0, 'w': 0.25	200	0.9094384144982766
'c1': 1.0, 'c2': 0.5, 'w': 0.15	10	1.002623410047945
'c1': 0.5, 'c2': 1.0, 'w': 0.25	10	1.0042683461220794

- The best solution is (options: ['c1': 1.0, 'c2': 0.5, 'w': 0.15], number of particles: 100), and result is 0.0020883614348142787
- This experiment took 49 seconds to optimize the mathematical functions.
- The options ['c1': 1.0, 'c2': 0.5, 'w': 0.15] is the best as they are used by the best solution, and used by other 2 solutions in the table.
- In terms of number of particles, the best value is 100, since the best solution use this value, and the result is much smaller than other solutions.

4.4.4 Comparative Analysis

Based on the above three algorithms optimizing the Griewank function, following conclusions can be made:

- Differential evolution has the best performance compared with the other two algorithms, as all of the top 10 solutions find the global minimum 0.0, while genetic algorithm gets the lowest cost 4.353184479555239e-13. Particle swarm optimization does not perform well as its best solution's cost is 0.0020883614348142787.
- Genetic algorithm may have better performance when increasing its number of generations and population size. By this way, the result may be closer to global minimum.
- All of the three algorithms takes less than 1 minute. Particle swarm optimization takes more time (49s) than genetic algorithm (19s) and differential evolution (28s).

4.5 Schwefel Function

In experiments, a list of floats will be used. The maximum of the number is 5, and the minimum number is -5. The number of evaluations for the three algorithms is 2500.

4.5.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used. Both of the number of generations and the population size are 50.

The table 4.10 are the top 10 optimization solutions in the 64 solutions.

Table 4.10: Genetic Algorithm on Schwefel Function

Crossover Rate	Mutation Rate	Result
0.2	0.4	415.0375983747156
0.3	0.01	415.0375983747156
0.3	0.4	415.0375983747156
0.3	0.9	415.0375983747156
0.5	0.01	415.0375983747156
0.5	0.05	415.0375983747156
0.5	0.1	415.0375983747156
0.5	0.25	415.0375983747156
0.5	0.4	415.0375983747156
0.5	0.6	415.0375983747156

From the table 4.10, the following conclusion can be made:

- All of the solutions find the local minimum 415.0375983747156.
- This experiment took 5 seconds to optimize the mathematical functions. As the number of evaluations get lower, the running time becomes shorter.

4.5.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize library. Both of the number of iterations and the population size are 50.

The table 4.11 are the top 10 optimization solutions in the 64 solutions.
From the table 4.11, the following conclusion can be made:

- All of the top 10 solutions get the cost 415.0491543422639.
- Differential evolution struggles in the local optima, not find the global optima.
- This experiment took 1 second to optimize the mathematical functions. As the number of evaluations get lower, the running time becomes shorter.

Table 4.11: Differential Evolution on Schwefel Function

Tolerance	Recombination	Mutation	result
0.01	0.5	(0.5, 1)	415.0491543422639
0.01	0.5	(0.5, 1.5)	415.0491543422639
0.01	0.5	(0.75, 1)	415.0491543422639
0.01	0.5	(0.75, 1.5)	415.0491543422639
0.01	0.6	(0.5, 1)	415.0491543422639
0.01	0.6	(0.5, 1.5)	415.0491543422639
0.01	0.6	(0.75, 1)	415.0491543422639
0.01	0.6	(0.75, 1.5)	415.0491543422639
0.01	0.8	(0.5, 1)	415.0491543422639
0.01	0.8	(0.5, 1.5)	415.0491543422639

4.5.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms. In this experiment, the list of number of particles is set to [1, 10, 25, 50, 100, 125] with 2500 times evaluations.

The table 4.12 are the top 10 optimization solutions in the 36 solutions.

Table 4.12: Particle Swarm Optimization on Schwefel Function

Options	Particles	result
'c1': 0.5, 'c2': 1.0, 'w': 0.15	1	415.0971569332407
'c1': 1.0, 'c2': 0.5, 'w': 0.15	1	415.5521192677187
'c1': 1.0, 'c2': 1.0, 'w': 0.25	1	416.9394823821286
'c1': 1.0, 'c2': 1.0, 'w': 0.15	1	419.9963110968742
'c1': 0.5, 'c2': 0.5, 'w': 0.15	1	420.12976206145163
'c1': 0.5, 'c2': 1.0, 'w': 0.25	1	420.5533457292883
'c1': 0.5, 'c2': 1.0, 'w': 0.15	10	4154.948081969748
'c1': 1.0, 'c2': 1.0, 'w': 0.15	10	4172.393474673918
'c1': 0.5, 'c2': 0.5, 'w': 0.15	10	4182.407971259541
'c1': 1.0, 'c2': 1.0, 'w': 0.25	10	4186.712516556092

From the table 4.12, the following conclusion can be made:

- The best solution is (options: ['c1': 0.5, 'c2': 1.0, 'w': 0.15], number of particles: 1), and result is 415.0971569332407
- This experiment took 1 minute 12 seconds to optimize the mathematical functions.

- The best options are ['c1': 0.5, 'c2': 1.0, 'w': 0.15], as it is used 2 times in the top 10 solutions, and used by the best solution.
- In terms of number of particles, the best value is 1, all of the top 6 solutions are using this value. The function may not need many particles, and small number of particles can get good optimization.

4.5.4 Comparative Analysis

Based on the above three algorithms optimizing the Schwefel function, following conclusions can be made:

- Genetic algorithm has the best performance compared with the other two algorithms, as its best solution's result is 415.0375983747156, while differential evolution gets the minimum cost 415.0491543422639 and particle swarm optimization's minimum cost is 415.0971569332407. Differential evolution performs better than particle swarm optimization.
- Both of the genetic algorithm and differential evolution find 10 same costs, respectively.
- Particle swarm optimization takes more time (72 seconds) than genetic algorithm (5 seconds) and differential evolution (1 second). One reason is that the particle swarm optimization uses the value 1 as the number of particles, as the number of evaluations is 2500, the iterations will be 2500, which costs more time.

4.6 Rastrigin Function

In experiments, a list of floats will be used. The maximum of the number is 5, and the minimum number is -5. The number of evaluations is 2500.

4.6.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used. Both of the number of generations and the population size are 50.

The table 4.13 are the top 10 optimization solutions in the 64 solutions.

From the table 4.13, the following conclusion can be made:

- All of the solutions find the global minimum 0.0.
- This experiment took 4 seconds to optimize the mathematical functions. As the number of evaluations get lower, the running time becomes shorter.
- Genetic algorithm performs well in optimizing Rastrigin function.

Table 4.13: Genetic Algorithm on Rastrigin Function

Crossover Rate	Mutation Rate	Result
0.3	0.75	0.0
0.5	0.01	0.0
0.5	0.05	0.0
0.5	0.1	0.0
0.5	0.25	0.0
0.5	0.4	0.0
0.5	0.6	0.0
0.6	0.01	0.0
0.6	0.05	0.0
0.6	0.1	0.0

4.6.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize library. Both of the number of iterations and the population size are 50.

The table 4.14 are the top 10 optimization solutions in the 64 solutions.

Table 4.14: Differential Evolution on Rastrigin Function

Tolerance	Recombination	Mutation	result
0.01	0.5	(0.5, 1)	0.0
0.01	0.5	(0.5, 1.5)	0.0
0.01	0.5	(0.75, 1)	0.0
0.01	0.5	(0.75, 1.5)	0.0
0.01	0.6	(0.5, 1)	0.0
0.01	0.6	(0.5, 1.5)	0.0
0.01	0.6	(0.75, 1)	0.0
0.01	0.6	(0.75, 1.5)	0.0
0.01	0.8	(0.5, 1)	0.0
0.01	0.8	(0.5, 1.5)	0.0

From the table 4.14, the following conclusion can be made:

- Differential evolution finds the global minimum value 0.0 .
- This experiment took 18 seconds to optimize the mathematical functions. When decreasing the number of evaluations, the running time becomes shorter.
- Differential evolution performs well in optimizing Rastrigin function.

4.6.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms. In this experiment, the list of number of particles is set to [1, 10, 25, 50, 100, 125] with 2500 times evaluations.

The table 4.15 are the top 10 optimization solutions in the 36 solutions.

Table 4.15: Particle Swarm Optimization on Rastrigin Function

Options	Particles	result
'c1': 0.5, 'c2': 0.5, 'w': 0.15	1	6.099412640602425
'c1': 0.5, 'c2': 1.0, 'w': 0.25	1	9.180829525144599
'c1': 1.0, 'c2': 1.0, 'w': 0.15	1	9.421923067487871
'c1': 1.0, 'c2': 0.5, 'w': 0.15	1	9.704880266430601
'c1': 1.0, 'c2': 1.0, 'w': 0.25	1	10.717158956895776
'c1': 0.5, 'c2': 1.0, 'w': 0.25	10	14.507773568454994
'c1': 0.5, 'c2': 1.0, 'w': 0.15	1	20.364677216500535
'c1': 1.0, 'c2': 1.0, 'w': 0.15	10	107.57737723988687
'c1': 0.5, 'c2': 0.5, 'w': 0.15	10	110.25045381078434
'c1': 0.5, 'c2': 0.5, 'w': 0.15	125	125.07743162628071

From the table 4.15, the following conclusion can be made:

- The best solution is (options: ['c1': 0.5, 'c2': 0.5, 'w': 0.15], number of particles: 1), and result is 6.099412640602425
- This experiment took 1 minute 15 seconds to optimize the mathematical functions.
- The best options are ['c1': 0.5, 'c2': 0.5, 'w': 0.15], as it appears 3 times in the top 10 solutions, and the best solution use it.
- In terms of number of particles, the best value is 1, as it is used 6 times by the top of 10 solutions.

4.6.4 Comparative Analysis

Based on the above three algorithms optimizing the Rastrigin function, following conclusions can be made:

- Genetic algorithm and differential evolution have the better performance compared with particle swarm optimization, as both of them find the global minimum 0.0, while particle swarm optimization's minimum cost is 6.099412640602425.

- Particle swarm optimization takes more time (1m15s) than genetic algorithm (4s) and particle swarm optimization (18s). One reason is that the particle swarm optimization uses the value 1 as the number of particles, as the number of evaluations is 2500, the iterations will be 2500, which costs more time.
- Compared with genetic algorithm and differential evolution, particle swarm optimization does not perform well, as its cost is over 6, which is far from the global minimum 0. One possible reason is that the parameters are not fully tested, and some potential good combinations are missing. Another reason is that the algorithm might not perform well in the function that contains many local minimum.

4.7 Sphere Function

In experiments, a list of floats will be used. The maximum of the number is 5, and the minimum number is -5.

4.7.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used.

The table 4.16 are the top 10 optimization solutions in the 64 solutions.

Table 4.16: Genetic Algorithm on Sphere Function

Crossover Rate	Mutation Rate	Result
1.0	0.01	3.030114887001745e-13
0.8	0.01	1.883622053811099e-12
1.0	0.05	3.718009373011e-12
0.8	0.05	7.51155752444439e-11
0.8	0.1	1.817228554148557e-10
1.0	0.1	8.555170291749524e-10
0.6	0.1	2.177585819685863e-09
1.0	0.25	1.1637257958399252e-63
0.6	0.05	4.764578624650968e-09
0.8	0.25	2.519067771027475e-08

From the table 4.16, the following conclusion can be made:

- The best solution is (crossover rate: 1.0, mutation rate: 0.01), and its result is 3.030114887001745e-13.
- This experiment took 22 seconds to optimize the mathematical functions.

- For the crossover rate, the value 1.0 performs best, as it appears 3 times in the top 10 solutions.
- For the mutation rate, the value 0.01 performs best as it is used by the top 2 solutions.
- Genetic algorithm is stuck at the local minimum.

4.7.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize library.

The table 4.17 are the top 10 optimization solutions in the 64 solutions.

Table 4.17: Differential Evolution on Sphere Function

Tolerance	Recombination	Mutation	result
0.01	0.5	(0.5, 1)	0.0
0.01	0.5	(0.5, 1.5)	0.0
0.01	0.5	(0.75, 1)	0.0
0.01	0.5	(0.75, 1.5)	0.0
0.01	0.6	(0.5, 1)	0.0
0.01	0.6	(0.5, 1.5)	0.0
0.01	0.6	(0.75, 1)	0.0
0.01	0.6	(0.75, 1.5)	0.0
0.01	0.8	(0.5, 1)	0.0
0.01	0.8	(0.5, 1.5)	0.0

From the table 4.17, the following conclusion can be made:

- Differential evolution finds the global minimum value 0.0 .
- This experiment took 47 seconds to optimize the mathematical functions.
- Differential evolution performs well in optimizing Sphere function.

4.7.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms.

The table 4.18 are the top 10 optimization solutions in the 36 solutions.

From the table 4.18, the following conclusion can be made:

- The best solution is (options: ['c1': 0.5, 'c2': 0.5, 'w': 0.15], number of particles: 25), and result is 0.24224510250460873.

Table 4.18: Particle Swarm Optimization on Sphere Function

Options	Particles	result
'c1': 0.5, 'c2': 0.5, 'w': 0.15	25	0.24224510250460873
'c1': 0.5, 'c2': 1.0, 'w': 0.25	25	2.562677388569827
'c1': 1.0, 'c2': 1.0, 'w': 0.15	80	5.713633117734652
'c1': 1.0, 'c2': 1.0, 'w': 0.15	10	10.925829034364229
'c1': 0.5, 'c2': 1.0, 'w': 0.25	10	11.841684422629331
'c1': 1.0, 'c2': 0.5, 'w': 0.15	10	17.450897851061352
'c1': 1.0, 'c2': 0.5, 'w': 0.15	25	24.929196796089364
'c1': 1.0, 'c2': 1.0, 'w': 0.25	50	42.537271005037745
'c1': 0.5, 'c2': 1.0, 'w': 0.15	25	47.58251834444716
'c1': 1.0, 'c2': 1.0, 'w': 0.25	10	59.5534292404688

- This experiment took 48 seconds to optimize the mathematical functions.
- For the options, the combination ['c1': 0.5, 'c2': 0.5, 'w': 0.15] performs best as it is used by the best solution, whose cost is much smaller than those of other solutions in the table.
- In terms of number of particles, the best value is 25, as it is used by 4 solutions of the top 10, and used by the best solution.

4.7.4 Comparative Analysis

Based on the above three algorithms optimizing the Sphere function, following conclusions can be made:

- Differential evolution has the best performance compared with the other two algorithms, as its best solution's result is the global minimum 0.0, while genetic algorithm gets the minimum cost 3.030114887001745e-13 and particle swarm optimization's minimum cost is 0.24224510250460873.
- From the perspective of the difference between the best solution to the 10th solution, the genetic algorithm has the biggest differences, which suggests that its solutions have more diversity. The minimum cost for the differential evolution is 3.030114887001745e-13, and the cost of the 10th solution is 2.519067771027475e-08, which is much larger than the cost of best solution as the difference of the magnitude is 5. For the differential evolution, the difference of the magnitudes between the best solution and the 10th solution is 0, and 3 for the particle swarm optimization.
- All of the experiments are used less than 1 minute. Particle swarm optimization takes more time (48 seconds) than genetic algorithm (22 seconds) and differential

evolution (47 seconds). The three algorithms are efficient in solving sphere function.

- Compared with genetic algorithm and differential evolution, particle swarm optimization does not perform well, as its cost is over 0.2, which is far from the costs of other two algorithms (close or equal to 0.0). One possible reason is that the parameters are not fully tested, and some potential good combinations are missing. Another reason is that the algorithm might not be good at optimizing this kind of math functions.

4.8 Sum Squares Function

In experiments, a list of floats will be used. The maximum of the number is 5, and the minimum number is -5.

4.8.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used.

The table 4.19 are the top 10 optimization solutions in the 64 solutions.

Table 4.19: Genetic Algorithm on Sum Squares

Crossover Rate	Mutation Rate	Result
1.0	0.01	3.710209728041692e-12
1.0	0.05	6.416881595598939e-11
0.8	0.1	1.0932493864072553e-10
0.8	0.01	1.4073584455323892e-10
0.8	0.05	2.2921336737101084e-10
1.0	0.1	2.47242353839222e-10
0.6	0.01	5.447464984450476e-10
0.6	0.1	2.9490823072244514e-09
0.6	0.05	4.1075267373727215e-09
1.0	0.25	5.531245205828194e-09

From the table 4.19, the following conclusion can be made:

- The best solution is (crossover rate: 1.0, mutation rate: 0.01), and its result is 3.710209728041692e-12.
- This experiment took 22 seconds to optimize the mathematical functions.
- For the crossover rate, the value 1.0 performs best, as it appears 4 times in the top 10 solutions, and is used by the best solution.

- For the mutation rate, the value 0.01 performs best as it appears 3 times in the top 10 solutions, and used by the best solution.
- The genetic algorithm performs well in optimizing sum squares function as its lowest cost is close to the global minimum 0.0.

4.8.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize library.

The table 4.20 are the top 10 optimization solutions in the 64 solutions.

Table 4.20: Differential Evolution on Sum Squares Function

Tolerance	Recombination	Mutation	result
0.01	0.5	(0.5, 1)	0.0
0.01	0.5	(0.5, 1.5)	0.0
0.01	0.5	(0.75, 1)	0.0
0.01	0.5	(0.75, 1.5)	0.0
0.01	0.6	(0.5, 1)	0.0
0.01	0.6	(0.5, 1.5)	0.0
0.01	0.6	(0.75, 1)	0.0
0.01	0.6	(0.75, 1.5)	0.0
0.01	0.8	(0.5, 1)	0.0
0.01	0.8	(0.5, 1.5)	0.0

From the table 4.20, the following conclusion can be made:

- Differential evolution finds the global minimum value 0.0 .
- This experiment took 48 seconds to optimize the mathematical functions.
- Differential evolution performs well in optimizing Sum Squares function.

4.8.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms.

The table 4.21 are the top 10 optimization solutions in the 36 solutions.

From the table 4.21, the following conclusion can be made:

- The best solution is (options: ['c1': 1.0, 'c2': 0.5, 'w': 0.15], number of particles: 10), and result is 0.23427535404976765.
- This experiment took 52 seconds to optimize the mathematical functions.

Table 4.21: Particle Swarm Optimization on Sum Squares Function

Options	Particles	result
'c1': 1.0, 'c2': 0.5, 'w': 0.15	10	0.23427535404976765
'c1': 0.5, 'c2': 1.0, 'w': 0.25	25	10.704058983966465
'c1': 0.5, 'c2': 0.5, 'w': 0.15	10	14.549838202332003
'c1': 1.0, 'c2': 1.0, 'w': 0.15	10	89.41300128173856
'c1': 0.5, 'c2': 1.0, 'w': 0.25	10	241.68612828790077
'c1': 0.5, 'c2': 1.0, 'w': 0.15	10	252.33152746859218
'c1': 1.0, 'c2': 1.0, 'w': 0.15	100	340.72323669212636
'c1': 1.0, 'c2': 1.0, 'w': 0.25	10	345.75482467299366
'c1': 0.5, 'c2': 1.0, 'w': 0.15	25	806.7266227138366
'c1': 0.5, 'c2': 0.5, 'w': 0.15	25	1319.8903623171843

- For the options, the combination ['c1': 1.0, 'c2': 0.5, 'w': 0.15] performs best as it is used by the best solution, which is much lower than other solutions.
- In terms of number of particles, the best value is 10, as it is used 6 times in the top 10 solutions. The function may not need many particles, and small number of particles can get good optimization.

4.8.4 Comparative Analysis

Based on the above three algorithms optimizing the Sum Squares function, following conclusions can be made:

- Differential evolution has the best performance compared with the other two algorithms, as its best solution's result is the global minimum 0.0, while genetic algorithm gets the minimum cost 3.710209728041692e-12 and particle swarm optimization's minimum cost is 0.23427535404976765
- From the perspective of the difference between the best solution to the 10th solution, the particle swarm optimization has the biggest differences, which suggests that its solutions have more diversity. The minimum cost for the particle swarm optimization is 0.23427535404976765, and the cost of the 10th solution is 1319.8903623171843, which is much larger than the cost of best solution as the difference of the magnitude is 5. For the genetic algorithm, the difference of the magnitudes between the best solution and the 10th solution is 3, and 0 for the differential evolution.
- All of the experiments are used less than 1 minute. Particle swarm optimization takes more time (52 seconds) than genetic algorithm (22 seconds) and differential evolution (48 seconds). The three algorithms are efficient in solving sum squares

function. One reason particle swarm optimization uses more time is that it may need more to time for convergence.

- Compared with genetic algorithm and differential evolution, particle swarm optimization does not perform well, as its cost is over 0.2, which is far from the costs of other two algorithms (close or equal to 0.0). One possible reason is that the parameters are not fully tested, and some potential good combinations are missing. Another reason is that the algorithm might not be designed correctly, and some circumstance are not considered, therefore it is stuck on the local minimum.

4.9 Trid Function

In experiments, a list of floats will be used. The maximum of the number is 5, and the minimum number is -5. The number of evaluations is 2500.

4.9.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used. Both of the number of generations and the population size are 50.

The table 4.22 are the top 10 optimization solutions in the 64 solutions.

Table 4.22: Genetic Algorithm on Trid Function

Crossover Rate	Mutation Rate	Result
0.5	0.01	0.0
0.5	0.05	0.0
0.5	0.1	0.0
0.5	0.25	0.0
0.5	0.6	0.0
0.5	0.75	0.0
0.5	0.9	0.0
0.6	0.01	0.0
0.6	0.05	0.0
0.6	0.1	0.0

From the table 4.22, the following conclusion can be made:

- All of the solutions find the global minimum 0.0.
- This experiment took 7 seconds to optimize the mathematical functions. As the number of evaluations get lower, the running time becomes shorter.
- Genetic algorithm performs well in optimizing Trid function.

4.9.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize library. Both of the number of generations and the population size are 50.

The table 4.23 are the top 10 optimization solutions in the 64 solutions.

Table 4.23: Differential Evolution on Trid Function

Tolerance	Recombination	Mutation	result
0.01	0.5	(0.5, 1)	4.930380657631324e-32
0.01	0.5	(0.5, 1)	4.930380657631324e-32
0.01	0.8	(0.5, 1)	4.930380657631324e-32
0.01	0.8	(0.75, 1)	4.930380657631324e-32
0.01	1.0	(0.5, 1)	4.930380657631324e-32
0.01	1.0	(0.75, 1)	4.930380657631324e-32
0.02	0.5	(0.5, 1)	4.930380657631324e-32
0.02	0.6	(0.5, 1)	4.930380657631324e-32
0.02	0.6	(0.75, 1)	4.930380657631324e-32
0.02	0.8	(0.5, 1)	4.930380657631324e-32

From the table 4.23, the following conclusion can be made:

- All of the top 10 differential evolution solutions find the local minimum value 4.930380657631324e-32 .
- This experiment took 24 seconds to optimize the mathematical functions.
- Differential evolution performs well in optimizing Trid function.

4.9.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms. In this experiment, the list of number of particles is set to [1, 10, 25, 50, 100, 125] with 2500 times evaluations.

The table 4.24 are the top 10 optimization solutions in the 36 solutions.

From the table 4.24, the following conclusion can be made:

- The best solution is (options: ['c1': 0.5, 'c2': 1.0, 'w': 0.25], number of particles: 10), and result is 3.629615727666212.
- This experiment took 1 minute 14 seconds to optimize the mathematical functions.
- For the options, the combination ['c1': 0.5, 'c2': 1.0, 'w': 0.25] performs best as it appears 3 times in the top 10 solutions, and is used by the best solution.

Table 4.24: Particle Swarm Optimization on Trid Function

Options	Particles	result
'c1': 0.5, 'c2': 1.0, 'w': 0.25	10	3.629615727666212
'c1': 0.5, 'c2': 0.5, 'w': 0.15	1	4.453272145408291
'c1': 1.0, 'c2': 0.5, 'w': 0.15	1	6.5148433211452295
'c1': 1.0, 'c2': 1.0, 'w': 0.15	1	7.544526338517894
'c1': 0.5, 'c2': 1.0, 'w': 0.25	125	7.94804790059726
'c1': 1.0, 'c2': 1.0, 'w': 0.15	10	8.993161789681755
'c1': 1.0, 'c2': 1.0, 'w': 0.25	1	10.64432866735462
'c1': 0.5, 'c2': 1.0, 'w': 0.25	100	10.699764166511898
'c1': 0.5, 'c2': 1.0, 'w': 0.15	1	10.921134103004722
'c1': 1.0, 'c2': 1.0, 'w': 0.25	10	13.030768016524744

- In terms of number of particles, the value 10 performs best as it appears 2 times in the top 10 solutions.

4.9.4 Comparative Analysis

Based on the above three algorithms optimizing the Trid function, following conclusions can be made:

- Genetic algorithm has the best performance compared with the other two algorithms, as its best solution's result is the global minimum 0.0, while differential evolution gets the local minimum cost 4.930380657631324e-32 and differential evolution's minimum cost is 3.629615727666212.
- From the perspective of the difference between the best solution to the 10th solution, the particle swarm optimization has the biggest differences, which suggests that its solutions have more diversity. The minimum cost for the particle swarm optimization is 3.629615727666212, and the cost of the 10th solution is 13.030768016524744, which is larger than the cost of best solution. For both of the genetic algorithm and differential evolution, the differences of the magnitudes are 0.
- Particle swarm optimization takes more time (1 minute 14 seconds) than genetic algorithm (7 seconds) and differential evolution (24 seconds). One reason is that the particle swarm optimization uses the value 1 as the number of particles, as the number of evaluations is 2500, the iterations will be 2500, which costs more time.
- Compared with genetic algorithm and differential evolution, particle swarm optimization does not perform well, as its cost is over 3, which is far from the global minimum 0. One possible reason is that the parameters are not fully

tested, and some potential good combinations are missing. Another reason is that the algorithm might not perform well in the function that contains many local minimum.

4.10 Booth Function

In experiments, a list of floats will be used. The maximum of the number is 10, and the minimum number is -10.

4.10.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used.

The table 4.25 are the top 10 optimization solutions in the 64 solutions.

Table 4.25: Genetic Algorithm on Booth Function

Crossover Rate	Mutation Rate	Result
0.5	0.01	0.0
0.5	0.25	0.0
0.6	0.01	0.0
0.6	0.05	0.0
0.6	0.25	0.0
0.8	0.01	0.0
0.8	0.05	0.0
0.8	0.1	0.0
0.8	0.4	0.0
1.0	0.01	0.0

From the table 4.25, the following conclusion can be made:

- All of the top solutions find the global minimum 0.0.
- This experiment took 13 seconds to optimize the mathematical functions.
- Genetic algorithm performs well in optimizing Booth function.

4.10.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize library.

The table 4.26 are the top 10 optimization solutions in the 64 solutions.

From the table 4.26, the following conclusion can be made:

Table 4.26: Differential Evolution on Booth Function

Tolerance	Recombination	Mutation	result
0.01	0.6	(0.5, 1)	3.1554436208840472e-30
0.01	0.8	(0.5, 1)	3.1554436208840472e-30
0.01	1.0	(0.5, 1)	3.1554436208840472e-30
0.02	0.5	(0.5, 1)	3.1554436208840472e-30
0.02	0.8	(0.5, 1)	3.1554436208840472e-30
0.02	1.0	(0.5, 1)	3.1554436208840472e-30
0.05	0.6	(0.5, 1)	3.1554436208840472e-30
0.05	0.8	(0.5, 1)	3.1554436208840472e-30
0.05	1.0	(0.5, 1)	3.1554436208840472e-30
0.1	0.8	(0.5, 1)	3.1554436208840472e-30

- All of the top 10 solutions find the same cost, 3.1554436208840472e-30, while it is not the global minimum 0.0
- This experiment took 2 minutes 11 seconds to optimize the mathematical functions.
- Differential evolution performs well in optimizing Booth function.

4.10.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms. The table 4.27 are the top 10 optimization solutions in the 36 solutions.

Table 4.27: Particle Swarm Optimization on Booth Function

Options	Particles	result
'c1': 1.0, 'c2': 1.0, 'w': 0.15	50	0.0
'c1': 1.0, 'c2': 1.0, 'w': 0.15	80	0.0
'c1': 1.0, 'c2': 1.0, 'w': 0.15	100	0.0
'c1': 1.0, 'c2': 1.0, 'w': 0.15	200	0.0
'c1': 0.5, 'c2': 1.0, 'w': 0.15	50	0.0
'c1': 0.5, 'c2': 1.0, 'w': 0.15	80	0.0
'c1': 0.5, 'c2': 1.0, 'w': 0.15	100	0.0
'c1': 0.5, 'c2': 1.0, 'w': 0.15	200	0.0
'c1': 0.5, 'c2': 1.0, 'w': 0.15	25	0.0
'c1': 0.5, 'c2': 1.0, 'w': 0.15	50	0.0

From the table 4.27, the following conclusion can be made:

- All of the top solutions find the global minimum 0.0.
- This experiment took 44 seconds to optimize the mathematical functions.
- Particle swarm optimization algorithms has great performance in optimizing Booth function.

4.10.4 Comparative Analysis

Based on the above three algorithms optimizing the Booth function, following conclusions can be made:

- Genetic algorithm and particle swarm optimization have better performance than the differential evolution, as both of them find the global minimum 0.0. Differential evolution finds the local minimum 3.1554436208840472e-30, which is close to 0.0.
- Differential evolution took over 2 minutes to optimize the Booth function, and genetic algorithm took 11 seconds and particle swarm optimization spent 44 seconds. The reason that the running time of differential evolution is larger than those of two other algorithms may be the slower speed of converge.
- Particle swarm optimization is likely to have good performance in optimizing polynomial function such as Booth function.

4.11 Michalewicz Function

In experiments, a list of floats will be used. The maximum of the number is π, and the minimum number is 0.

4.11.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used.

The table 4.28 are the top 10 optimization solutions in the 64 solutions.

From the table 4.28, the following conclusion can be made:

- The best solution is (crossover rate: 0.5, mutation rate: 0.1), and its result is -9.493596164656133.
- This experiment took 33 seconds to optimize the mathematical functions.
- For the crossover rate, the value 0.2 performs well in optimizing Michalewicz function, as it appears twice in the table and is used by the best solution.
- For the mutation rate, the value 0.25 performs best as it is used 4 times in the top 10 solutions and is also used by the best solution.

Table 4.28: Genetic Algorithm on Michalewicz Function

Crossover Rate	Mutation Rate	Result
0.2	0.25	-9.493596164656133
0.5	0.4	-9.477464347516236
1.0	0.1	-9.374980490980324
1.0	0.05	-9.338645322834303
0.1	0.1	-9.285550601038626
0.8	0.25	-9.265343202243724
0.2	0.4	-9.252249514173322
0.5	0.25	-9.191200634459266
0.3	0.4	-9.146671882124716
0.6	0.25	-9.14263293173073

4.11.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize library.

The table 4.29 are the top 10 optimization solutions in the 64 solutions.

Table 4.29: Differential Evolution on Michalewicz Function

Tolerance	Recombination	Mutation	result
0.02	1.0	(0.75, 1.5)	-0.9678506611007817
0.01	1.0	(0.75, 1.5)	-0.967850661100781
0.05	0.8	(0.5, 1.5)	-0.9678506611007807
0.01	0.8	(0.75, 1)	-0.9678506611007803
0.1	1.0	(0.75, 1)	-0.9678506611007801
0.02	1.0	(0.5, 1)	-0.96785066110078
0.05	1.0	(0.75, 1.5)	-0.96785066110078
0.1	0.6	(0.5, 1)	-0.96785066110078
0.05	0.5	(0.5, 1)	-0.9678506611007799
0.05	0.6	(0.5, 1.5)	-0.9678506611007799

From the table 4.29, the following conclusion can be made:

- The best solution is (tolerance: 0.02, recombination: 1.0, mutation: (0.75, 1)), and result is -0.9678506611007817
- This experiment took 10 seconds to optimize the mathematical functions.
- For the tolerance, the value 0.02 perform well as it appears 2 times in the top 10 solutions and is used by the best solution.

- In terms of recombination rate, 1.0 is the best value of the test list, since it is used 5 times in the top 10 solutions, and also used by the best solution.
- For the mutation tuple, (0.75, 1) is the best, as it appears 5 times in the top 10 solutions.

4.11.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms.

The table 4.30 are the top 10 optimization solutions in the 36 solutions.

Table 4.30: Particle Swarm Optimization on Michalewicz Function

Options	Particles	result
'c1': 0.5, 'c2': 1.0, 'w': 0.15	200	-43.92854152286621
'c1': 0.5, 'c2': 1.0, 'w': 0.25	200	-38.253407500295886
'c1': 1.0, 'c2': 1.0, 'w': 0.25	200	-33.17175600518572
'c1': 1.0, 'c2': 1.0, 'w': 0.15	100	-26.21333308443039
'c1': 1.0, 'c2': 0.5, 'w': 0.15	200	-18.006351777650274
'c1': 1.0, 'c2': 0.5, 'w': 0.15	80	-16.134521909711367
'c1': 0.5, 'c2': 1.0, 'w': 0.25	80	-15.000291917071706
'c1': 0.5, 'c2': 1.0, 'w': 0.25	50	-10.44864439425058
'c1': 1.0, 'c2': 0.5, 'w': 0.15	25	-7.612207475217915
'c1': 0.5, 'c2': 1.0, 'w': 0.25	25	-6.83197557815739

From the table 4.30, the following conclusion can be made:

- The best solution is (options: ['c1': 0.5, 'c2': 1.0, 'w': 0.15], number of particles: 200), and result is -43.92854152286621.
- This experiment took 58 seconds to optimize the mathematical functions.
- For the options, the combination ['c1': 0.5, 'c2': 1.0, 'w': 0.15] performs best as it is used by the best solution.
- In terms of number of particles, the value 200 performs best as it is used by all of the solutions.

4.11.4 Comparative Analysis

Based on the above three algorithms optimizing the Michalewicz function, following conclusions can be made:

- For the Michalewicz function, the particle swarm optimization gets the minimum value -43.92854152286621, which is smaller than minimum costs of genetic algorithm and differential evolution. Particle swarm optimization has the best performance.
- Particle swarm optimization takes more time (58 seconds) than genetic algorithm (33 seconds) and differential evolution (10 seconds). One reason is that the convergence speed of particle swarm optimization may be slower than the other two algorithms.

4.12 Matyas Function

In experiments, a list of floats will be used. The maximum of the number is 10, and the minimum number is -10.

4.12.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used.

The table 4.31 are the top 10 optimization solutions in the 64 solutions.

Table 4.31: Genetic Algorithm on Matyas Function

Crossover Rate	Mutation Rate	Result
0.6	0.25	3.939349307768404e-27
0.6	0.05	6.403016067294892e-25
0.5	0.75	1.3449232018213675e-24
0.6	0.6	9.272315756837494e-18
0.3	0.75	5.109031439380338e-16
0.6	0.01	3.0137859818180668e-15
0.6	0.75	1.337450530332821e-13
0.5	0.1	2.3376344617034664e-12
0.8	0.05	2.582685216570985e-12
0.8	0.25	2.1142826222669738e-11

From the table 4.31, the following conclusion can be made:

- The best solution is (crossover rate: 0.6, mutation rate: 0.01), and its result is 3.939349307768404e-27.
- This experiment took 13 seconds to optimize the mathematical functions.
- For the crossover rate, the value 0.6 perform best, as it appears 3 times in the top 4 solutions.

- From the perspective of mutation rate, the value 0.25 has the best performance as it appears 2 times in the table and is used by the best solution.

4.12.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize library.

The table 4.32 are the top 10 optimization solutions in the 64 solutions.

Table 4.32: Differential Evolution on Matyas Function

Tolerance	Recombination	Mutation	result
0.01	0.8	(0.5, 1)	0.0
0.01	1.0	(0.5, 1)	0.0
0.02	0.8	(0.5, 1)	0.0
0.02	1.0	(0.5, 1)	0.0
0.05	0.8	(0.5, 1)	0.0
0.05	1.0	(0.5, 1)	0.0
0.1	0.8	(0.5, 1)	0.0
0.1	1.0	(0.5, 1)	0.0
0.1	1.0	(0.5, 1.5)	0.0
0.1	1.0	(0.75, 1)	3.2047474274603605e-31

From the table 4.32, the following conclusion can be made:

- All of the top 9 solutions find the global minimum 0.0.
- This experiment took 2 minutes 25 seconds to optimize the mathematical functions.
- Differential evolution performs well in optimizing Matyas function.

4.12.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms.

The table 4.33 are the top 10 optimization solutions in the 36 solutions.

From the table 4.33, the following conclusion can be made:

- The best solution is (options: ['c1': 1.0, 'c2': 1.0, 'w': 0.25], number of particles: 25), and result is 6.949750915536675e-126.
- This experiment took 48 seconds to optimize the mathematical functions.
- For the options, the combination ['c1': 1.0, 'c2': 1.0, 'w': 0.25] performs best as it appears 3 times in the top 10 solutions, and is used by the best solution.

Table 4.33: Particle Swarm Optimization on Matyas Function

Options	Particles	result
'c1': 1.0, 'c2': 1.0, 'w': 0.25	25	6.949750915536675e-126
'c1': 0.5, 'c2': 1.0, 'w': 0.25	50	1.9382155213351847e-98
'c1': 1.0, 'c2': 1.0, 'w': 0.25	50	2.217945130617099e-86
'c1': 0.5, 'c2': 1.0, 'w': 0.15	80	8.85012380774194e-76
'c1': 1.0, 'c2': 1.0, 'w': 0.15	80	9.044027075063828e-72
'c1': 1.0, 'c2': 1.0, 'w': 0.15	50	8.674808520832527e-70
'c1': 1.0, 'c2': 0.5, 'w': 0.15	50	2.9447805039468112e-64
'c1': 0.5, 'c2': 1.0, 'w': 0.15	100	2.1079097281763427e-63
'c1': 0.5, 'c2': 1.0, 'w': 0.25	80	1.301878393928711e-62
'c1': 1.0, 'c2': 1.0, 'w': 0.25	80	6.728370793067629e-59

- In terms of number of particles, the value 25 performs best as it used by the best solution. The value 50 also performs well as it appears 4 times in the top 10 solutions.

4.12.4 Comparative Analysis

Based on the above three algorithms optimizing the Matyas function, following conclusions can be made:

- Differential evolution has better performance than the genetic algorithm and particle swarm optimization, as it finds the global minimum 0.0. Genetic algorithm finds the local minimum 3.939349307768404e-27, and particle swarm optimization's minimum cost is 6.949750915536675e-126.
- The minimum costs of genetic algorithm and particle swarm optimization are close to global minimum 0.0.
- Differential evolution took 2 minutes 25 seconds to optimize the Matyas function, and genetic algorithm took 13 seconds and particle swarm optimization spent 48 seconds. One reason that it takes differential evolution more time to optimize is increasing the number of inputs.
- Particle swarm optimization is likely to have good performance in optimizing polynomial function such as Matyas function, which is similar to the Booth function.

4.13 McCormick Function

In experiments, a list of floats will be used. The maximum of the number is 10, and the minimum number is -10.

4.13.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used.

The table 4.34 are the top 10 optimization solutions in the 56 solutions.

Table 4.34: Genetic Algorithm on McCormick Function

Crossover Rate	Mutation Rate	Result
0.5	0.01	-23.904371530109593
0.3	0.9	-23.90427303244148
0.6	0.9	-21.87103061172818
0.6	0.05	-20.762778876519803
0.3	0.75	-20.762778714322984
0.5	0.05	-20.762747409063113
0.5	0.25	-17.62118622293001
0.5	0.75	-17.62118622293001
0.5	0.9	-17.62118622293001
0.8	0.25	-17.62118622293001

From the table 4.34, the following conclusion can be made:

- The best solution is (crossover rate: 0.8, mutation rate: 0.25), and its result is -23.904371530109593.
- This experiment took 14 seconds to optimize the mathematical functions.
- For the crossover rate, the value 0.5 perform best, as it appears 5 times in the top 10 solutions.
- For the mutation rate, the best solution uses the value 0.01. The value 0.9 also performs well as it appears 3 times in the top 10 solutions.

4.13.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize library.

The table 4.35 are the top 10 optimization solutions in the 64 solutions.

From the table 4.35, the following conclusion can be made:

- The best solutions are (tolerance: 0.01, recombination: 0.8, mutation: (0.75, 1)) and (tolerance: 0.1, recombination: 0.6, mutation: (0.5, 1)), and both of them get the result is -10.122147076027831.
- This experiment took 34 seconds to optimize the mathematical functions.

Table 4.35: Differential Evolution on McCormick Function

Tolerance	Recombination	Mutation	result
0.01	0.8	(0.75, 1.5)	-10.122147076027831
0.1	0.6	(0.5, 1)	-10.122147076027831
0.01	0.6	(0.75, 1.5)	-10.12214707602783
0.01	0.8	(0.5, 1)	-10.12214707602783
0.01	0.8	(0.75, 1)	-10.12214707602783
0.02	0.5	(0.5, 1)	-10.12214707602783
0.05	0.5	(0.75, 1.5)	-10.12214707602783
0.05	0.6	(0.75, 1)	-10.12214707602783
0.05	0.8	(0.75, 1.5)	-10.12214707602783
0.05	1.0	(0.5, 1)	-10.12214707602783

- For the tolerance, the value 0.01 perform well as it appears 4 times in the top 5 solutions and is used by the best solution.
- In terms of recombination rate, 0.8 is the best value of the test list, since it is used 3 times in the top 5 solutions, and also used by the best solution.
- For the mutation tuple, (0.75, 1.5) is the best, as it appears 4 times in the top 10 solutions.

4.13.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms. The table 4.36 are the top 10 optimization solutions in the 36 solutions.

Table 4.36: Particle Swarm Optimization on McCormick Function

Options	Particles	result
'c1': 0.5, 'c2': 1.0, 'w': 0.25	25	-10.121772395404127
'c1': 1.0, 'c2': 1.0, 'w': 0.15	100	-10.119899370091684
'c1': 0.5, 'c2': 1.0, 'w': 0.25	200	-10.119310391176946
'c1': 0.5, 'c2': 1.0, 'w': 0.15	100	-10.119258530575598
'c1': 0.5, 'c2': 1.0, 'w': 0.25	100	-10.118387406168342
'c1': 1.0, 'c2': 1.0, 'w': 0.25	80	-10.112919312731826
'c1': 1.0, 'c2': 1.0, 'w': 0.25	200	-10.111745948327751
'c1': 0.5, 'c2': 1.0, 'w': 0.25	80	-10.111635156854895
'c1': 1.0, 'c2': 1.0, 'w': 0.25	100	-10.095610400580298
'c1': 1.0, 'c2': 1.0, 'w': 0.25	25	-10.085753367605086

From the table 4.36, the following conclusion can be made:

- The best solution is (options: ['c1': 0.5, 'c2': 1.0, 'w': 0.25], number of particles: 25), and result is -10.121772395404127.
- This experiment took 49 seconds to optimize the mathematical functions.
- For the options, the combination ['c1': 0.5, 'c2': 1.0, 'w': 0.25] performs best as it appears 4 times in the top 10 solutions and is used by the best solution.
- In terms of number of particles, the value 25 performs best as it is used by 2 solutions out of 10 in the table, and also used by the best solution. The value 100 also has good performance as it appears 4 times in the table.

4.13.4 Comparative Analysis

Based on the above three algorithms optimizing the McCormick function, following conclusions can be made:

- Genetic algorithm has the best performance compared with the other two algorithms, as its best solution's result is -23.904371530109593, while differential evolution gets the minimum cost -10.122147076027831 and particle swarm optimization's minimum cost is -10.121772395404127.
- The minimum costs of differential evolution and particle swarm optimization is close, but still have distance to the cost of genetic algorithm. Differential evolution and particle swarm optimization are stuck in the local minimum, and need tuning parameters to find the global minimum.
- Differential evolution took around 34 seconds to optimize the McCormick function, and genetic algorithm took 14 seconds and particle swarm optimization spent 49 seconds. The three algorithms are efficient in solving McCormick function.

4.14 Cross-in-tray Function

In experiments, a list of floats will be used. The maximum of the number is 10, and the minimum number is -10.

4.14.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used.

The table 4.37 are the top 10 optimization solutions in the 64 solutions.

From the table 4.37, the following conclusion can be made:

- All of the top 10 solutions find the same local minimum -2.0626118708227397, which is close to the global minimum -2.06261

Table 4.37: Genetic Algorithm on Cross-in-tray Function

Crossover Rate	Mutation Rate	Result
0.2	0.9	-2.0626118708227397
0.3	0.1	-2.0626118708227397
0.3	0.6	-2.0626118708227397
0.3	0.75	-2.0626118708227397
0.5	0.01	-2.0626118708227397
0.5	0.05	-2.0626118708227397
0.5	0.1	-2.0626118708227397
0.5	0.25	-2.0626118708227397
0.5	0.4	-2.0626118708227397
0.5	0.6	-2.0626118708227397

- This experiment took 14 seconds to optimize the mathematical functions.
- The difference between the local minimum and global minimum is 9.070171965789129e-05%
- Genetic algorithm performs well in optimizing the cross-in-tray function.

4.14.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize library.

The table 4.38 are the top 10 optimization solutions in the 64 solutions.

Table 4.38: Differential Evolution on Cross-in-tray Function

Tolerance	Recombination	Mutation	result
0.1	0.6	(0.5, 1)	-2.0626118708227397
0.02	0.5	(0.5, 1.5)	-2.062611870822724
0.1	0.8	(0.5, 1)	-2.0626118708226975
0.01	1.0	(0.5, 1.5)	-2.062611870822677
0.02	0.5	(0.75, 1)	-2.0626118708226606
0.1	0.6	(0.75, 1)	-2.062611870822638
0.1	0.8	(0.75, 1)	-2.0626118708224914
0.02	0.6	(0.75, 1)	-2.0626118708220025
0.01	0.8	(0.75, 1.5)	-2.062611870821999
0.05	0.6	(0.75, 1.5)	-2.062611870821986

From the table 4.38, the following conclusion can be made:

- The best solution is (tolerance: 0.1, recombination: 0.6, mutation: (0.5, 1)), and result is -2.0626118708227397
- This experiment took 14 seconds to optimize the mathematical functions.
- The difference between the local minimum and global minimum is 9.070171965789129e-05%
- For the tolerance, the value 0.1 performs well as it appears 4 times in the table, and is used by the best solution.
- In terms of recombination rate, 0.6 is the best value of the test list, since it appears 4 times in the table, and is used by the best solution.
- For the mutation tuple, (0.5, 1) is the best, as it appears twice times in the top 3 solutions.
- Differential evolution performs well in optimizing the cross-in-tray function.

4.14.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms.
The table 4.39 are the top 10 optimization solutions in the 36 solutions.

Table 4.39: Particle Swarm Optimization on Cross-in-tray Function

Options	Particles	result
'c1': 1.0, 'c2': 1.0, 'w': 0.15	50	-2.0626118708227397
'c1': 1.0, 'c2': 1.0, 'w': 0.15	80	-2.0626118708227397
'c1': 1.0, 'c2': 1.0, 'w': 0.15	100	-2.0626118708227397
'c1': 1.0, 'c2': 1.0, 'w': 0.15	200	-2.0626118708227397
'c1': 1.0, 'c2': 0.5, 'w': 0.15	100	-2.0626118708227397
'c1': 1.0, 'c2': 0.5, 'w': 0.15	20	-2.0626118708227397
'c1': 0.5, 'c2': 1.0, 'w': 0.15	50	-2.0626118708227397
'c1': 0.5, 'c2': 1.0, 'w': 0.15	80	-2.0626118708227397
'c1': 0.5, 'c2': 1.0, 'w': 0.15	100	-2.0626118708227397
'c1': 0.5, 'c2': 1.0, 'w': 0.15	200	-2.0626118708227397

From the table 4.39, the following conclusion can be made:

- All of the top 10 solutions find the same local minimum -2.0626118708227397, which is close to the global minimum -2.06261
- This experiment took 46 seconds to optimize the mathematical functions.

- The difference between the local minimum and global minimum is 9.070171965789129e-05%
- Particle swarm optimization performs well in optimizing the cross-in-tray function.

4.14.4 Comparative Analysis

Based on the above three algorithms optimizing the cross-in-tray function, following conclusions can be made:

- All of the three algorithm get the same local minimum -2.0626118708227397, which is close to the global minimum -2.06261
- Genetic algorithm, differential evolution and particle swarm optimization have good performance in optimizing cross-in-tray function.
- Differential evolution took 14 seconds to optimize the cross-in-tray function, and genetic algorithm took 14 seconds and particle swarm optimization spent 46 seconds. All of them are efficient to solve this function.

4.15 Drop Wave Function

In experiments, a list of floats will be used. The maximum of the number is 5.12, and the minimum number is -5.12.

4.15.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used.

The table 4.40 are the top 10 optimization solutions in the 64 solutions.

From the table 4.40, the following conclusion can be made:

- The top 3 solutions find the global minimum -1.0.
- This experiment took 13 seconds to optimize the mathematical functions.
- Genetic algorithm has great performance in optimizing drop wave function.
- For the crossover rate, both of the values 0.5 and 0.6 perform well, either of them appears in the top 4 solutions.
- For the mutation rate, the values 0.25, 0.05 and 0.1 have good performance as the top 3 solutions use one of them.

Table 4.40: Genetic Algorithm on Drop Wave Function

Crossover Rate	Mutation Rate	Result
0.5	0.25	-1.0
0.6	0.05	-1.0
0.6	0.1	-1.0
0.6	0.9	-0.9999999999739071
0.1	0.75	-0.9999999999153594
0.2	0.75	-0.999999999806197
0.1	0.4	-0.999999020119217
0.2	0.6	-0.9999941954435668
0.1	0.9	-0.9999933729342836
0.1	0.6	-0.9999583919763834

4.15.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize library.

The table 4.41 are the top 10 optimization solutions in the 64 solutions.

Table 4.41: Differential Evolution on Drop Wave Function

Tolerance	Recombination	Mutation	result
0.1	0.6	(0.75, 1.5)	-0.9999999999999996
0.01	0.8	(0.75, 1.5)	-0.9999999999999989
0.02	0.6	(0.75, 1.5)	-0.9999999999999989
0.1	0.8	(0.75, 1.5)	-0.9999999999999988
0.01	1.0	(0.75, 1)	-0.9999999999999987
0.02	1.0	(0.5, 1)	-0.9999999999999986
0.02	0.6	(0.75, 1)	-0.9999999999999984
0.05	0.5	(0.75, 1.5)	-0.9999999999999984
0.1	0.8	(0.5, 1.5)	-0.9999999999999984
0.01	0.5	(0.75, 1)	-0.9999999999999983

From the table 4.41, the following conclusion can be made:

- The best solution is (tolerance: 0.1, recombination: 0.6, mutation: (0.75, 1.5)), and result is -0.9999999999999996, which is close to the global minimum -1.0
- This experiment took 49 seconds to optimize the mathematical functions.
- For the tolerance, the value 0.1 performs well as it appears 3 times in the table, and is used by the best solution.

- In terms of recombination rate, 0.6 is the best value of the test list, since it appears 3 times in the table, and is used by the best solution.
- From the perspective of mutation tuple, (0.75, 1.5) performs best as all of the top 5 solutions use it.
- Differential evolution performs well in optimizing drop wave function.

4.15.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms.

The table 4.42 are the top 10 optimization solutions in the 36 solutions.

Table 4.42: Particle Swarm Optimization on Drop Wave Function

Options	Particles	result
'c1': 1.0, 'c2': 1.0, 'w': 0.15	25	-1.0
'c1': 1.0, 'c2': 1.0, 'w': 0.15	50	-1.0
'c1': 1.0, 'c2': 1.0, 'w': 0.15	100	-1.0
'c1': 1.0, 'c2': 1.0, 'w': 0.15	200	-1.0
'c1': 0.5, 'c2': 1.0, 'w': 0.15	50	-1.0
'c1': 0.5, 'c2': 1.0, 'w': 0.15	80	-1.0
'c1': 0.5, 'c2': 1.0, 'w': 0.15	100	-1.0
'c1': 0.5, 'c2': 1.0, 'w': 0.15	200	-1.0
'c1': 0.5, 'c2': 1.0, 'w': 0.25	80	-1.0
'c1': 0.5, 'c2': 1.0, 'w': 0.25	100	-1.0

From the table 4.42, the following conclusion can be made:

- All of the top 10 solutions find the global minimum -1.0.
- This experiment took 44 seconds to optimize the mathematical functions.
- Particle swarm optimization has great performance in optimizing drop wave function.

4.15.4 Comparative Analysis

Based on the above three algorithms optimizing the drop wave function, following conclusions can be made:

- Genetic algorithm and particle swarm optimization find the global minimum -1.0, and differential evolution gets the minimum cost -0.9999999999999996, which is close to the global minimum.

- For particle swarm optimization, all of the top 10 solutions find the global minimum, while genetic algorithm finds 3 solution with global minimum.
- Differential evolution took 49 seconds to optimize the drop wave function, and genetic algorithm took 13 seconds and particle swarm optimization spent 44 seconds.The three algorithms are efficient to optimize the function.

4.16 Holder Table Function

In experiments, a list of floats will be used. The maximum of the number is 10, and the minimum number is -10.

4.16.1 Genetic Algorithm

In the implementation of genetic algorithm, the library deap is used.

The table 4.43 are the top 10 optimization solutions in the 64 solutions.

Table 4.43: Genetic Algorithm on Holder Table Function

Crossover Rate	Mutation Rate	Result
0.6	0.25	-19.208502567886754
0.8	0.1	-19.208502567886754
0.8	0.25	-19.208502567886754
0.2	0.75	-19.20850256788675
0.5	0.01	-19.20850256788675
0.5	0.1	-19.20850256788675
0.5	0.25	-19.20850256788675
0.5	0.4	-19.20850256788675
0.5	0.75	-19.20850256788675
0.5	0.9	-19.20850256788675

From the table 4.43, the following conclusion can be made:

- The top 3 solutions find the local minimum -19.208502567886754, which is close to the global minimum -19.2085
- The difference between the local minimum and global minimum is 1.3368491829207e-05%
- This experiment took 21 seconds to optimize the mathematical functions.
- For the crossover rate, both of the values 0.5 and 0.6 perform well. One of the best solution uses 0.6 and 0.5 appears 5 times in the table.

- For the mutation rate, the values 0.25 has good performance as it appears 3 times in the top 10 solutions.
- Genetic algorithm performs well in optimizing the holder table function.

4.16.2 Differential Evolution

In the implementation of differential evolution, we are using differential_evolution from the scipy.optimize library.

The table 4.44 are the top 10 optimization solutions in the 64 solutions.

Table 4.44: Differential Evolution on Holder Table Function

Tolerance	Recombination	Mutation	result
0.02	0.6	(0.5, 1.5)	-19.208502567886736
0.02	0.8	(0.75, 1.5)	-19.208502567886732
0.05	0.8	(0.75, 1.5)	-19.208502567886732
0.05	1.0	(0.75, 1)	-19.208502567886732
0.1	0.8	(0.5, 1.5)	-19.208502567886732
0.05	0.6	(0.75, 1)	-19.20850256788673
0.01	0.5	(0.5, 1.5)	-19.208502567886725
0.01	0.8	(0.75, 1)	-19.208502567886725
0.02	0.6	(0.75, 1)	-19.20850256788672
0.05	0.8	(0.5, 1)	-19.20850256788672

From the table 4.44, the following conclusion can be made:

- The best solution is (tolerance: 0.02, recombination: 0.6, mutation: (0.5, 1.5)), and result is -19.208502567886736
- The difference between the local minimum and global minimum is 1.3368491736729354e-05%
- This experiment took 26 seconds to optimize the mathematical functions.
- For the tolerance, the value 0.02 performs well as it appears 3 times in the table, and is used by the best solution.
- In terms of recombination rate, 0.6 is the best value of the test list, since it appears 3 times in the table, and is used by the best solution.
- From the perspective of mutation tuple, (0.5, 1.5) performs best as it appears twice in the top 5 solutions.
- Differential evolution performs well in optimizing holder table function.

4.16.3 Particle Swarm Optimization

In the implementation of particle swarm optimization, we are using the library pyswarms.
The table 4.45 are the top 10 optimization solutions in the 36 solutions.

Table 4.45: Particle Swarm Optimization on Holder Table Function

Options	Particles	result
'c1': 1.0, 'c2': 1.0, 'w': 0.15	80	-19.208502567886754
'c1': 1.0, 'c2': 1.0, 'w': 0.15	100	-19.208502567886754
'c1': 0.5, 'c2': 1.0, 'w': 0.15	200	-19.208502567886754
'c1': 0.5, 'c2': 1.0, 'w': 0.25	50	-19.208502567886754
'c1': 0.5, 'c2': 1.0, 'w': 0.25	80	-19.208502567886754
'c1': 0.5, 'c2': 1.0, 'w': 0.25	100	-19.208502567886754
'c1': 1.0, 'c2': 1.0, 'w': 0.25	80	-19.208502567886754
'c1': 1.0, 'c2': 1.0, 'w': 0.25	100	-19.208502567886754
'c1': 1.0, 'c2': 1.0, 'w': 0.15	200	-19.20850256788675
'c1': 0.5, 'c2': 1.0, 'w': 0.15	80	-19.20850256788675

From the table 4.45, the following conclusion can be made:

- There are 8 solutions find the local minimum -19.208502567886754, which is close to the global minimum -19.2085
- This experiment took 44 seconds to optimize the mathematical functions.
- The difference between the local minimum and global minimum is 1.3368491829207e-05%
- For the options, the combination ['c1': 1.0, 'c2': 1.0, 'w': 0.15] performs best as it appears 3 times in the top 10 solutions and is used by the best solution.
- In terms of number of particles, the value 80 performs best as it is used by 3 solutions out of 10 in the table, and also used by the best solution. The value 100 also has good performance as it appears 3 times in the table.
- Particle swarm optimization performs well in optimizing holder table function.

4.16.4 Comparative Analysis

Based on the above three algorithms optimizing the holder table function, following conclusions can be made:

- Differential evolution has the best performance compared with the other two algorithms, as its best solution's result is -19.208502567886736, while both genetic algorithm and particle swarm optimization get the same minimum cost -19.208502567886754.

- All of the three algorithms have good performance in optimizing the holder table function, as their best solutions are close to the global minimum.
- Differential evolution took around 26 seconds to optimize the holder table function, and genetic algorithm took 21 seconds and particle swarm optimization spent 44 seconds. The three algorithms are efficient to optimize the function.

Chapter 5

Experiment Analysis

5.1 Interpretation of Results

5.1.1 Genetic Algorithm

Findings from the experiments of genetic algorithm:

- Regarding to the optimization on the 15 mathematical functions, genetic algorithm performs well in most of them. Genetic algorithm finds global minimum on optimizing Rastrigin function, Trid function, Booth function and drop wave function.
- Genetic algorithm is efficient. In the experiments, it generally takes no more than 1 minute.
- Crossover rate and mutation rate influence the optimization result significantly. Different functions may use totally different values of crossover rate and mutation rate. In terms of the crossover rate, the value 1.0 performs best in optimizing sum squares function, while the value 0.6 has good performance in optimizing drop wave function. From the perspective of mutation rate, 0.01 has the best performance in optimizing sphere function, while the largest value 0.9 in the test list performs best when optimizing Zakharov function.

5.1.2 Differential Evolution

Findings from the experiments of differential evolution:

- Differential evolution has good performance in optimizing most of the mathematical functions in the experiments. Differential evolution finds global minimum on optimizing sum of different powers function, Zakharov function, Griewank function, Rastrigin function, sphere function, sum squares function and Matyas function.

- Differential evolution is efficient. In the experiments, it generally takes no more than 1 minute.
- When the inputs array get larger, the running time becomes longer. For instance, it took more time to optimize Matyas function (2 minutes 25 seconds) than Rastrigin function (18 seconds).
- The three parameters tolerance, recombination and mutation play important roles in optimizing mathematical functions. Different functions may need varied tolerance, recombination and mutation.

5.1.3 Particle Swarm Optimization

Findings from the experiments of particle swarm optimization:

- Particle swarm optimization performs well in some mathematical functions, while not well in other functions. Particle swarm optimization finds the global minimum when solving the Booth function and drop wave function.
- Particle swarm optimization solves mathematical functions efficiently. In all of the experiments, particle swarm optimization takes no more than 3 minutes.
- The number of particles is important in optimization mathematical functions. In some mathematical functions such as Sum Squares function, the test value 10 has the best performance, while optimizing holder table function, the test value 80 performs well.

5.2 Comparison Among Algorithms

Based on the optimization on the 15 mathematical functions, we can compare the three algorithms, genetic algorithm, differential evolution and particle swarm optimization, which is displayed by the Table 5.1

From the table 5.1, differential evolution has 9 best solutions, which is more than genetic algorithm's 7 best solutions and particle swarm optimization's 4 best solutions. It can be concluded that differential evolution has better performance than the other two algorithms in optimizing the 15 mathematical functions.

From the perspective of running time, all of the experiments are less than 3 minutes, and most of them are less than 1 minute. For the genetic algorithm, the longest running time is optimizing the Michalewicz function with 33 seconds, and the shortest running time is solving the Rastrigin function, which only spent 4 seconds. For the differential evolution, all of the experiments' running time is under 1 minute. When using the particle swarm optimization, in some experiments, I tried using 1 particle with 2500 times evaluation, which means the iterations would be 2500, and this led to longer running time. For example, it took 75 seconds to optimize the Rastrigin function.

Table 5.1: Best Solution to the 15 Mathematical Functions

Function	Best Algorithm	result
Sum of Different Powers	Differential Evolution	0.0
Zakharov	Differential Evolution	0.0
Griewank	Differential Evolution	0.0
Schwefel	Genetic Algorithm	415.0375983747156
Rastrigin	Genetic Algorithm / Differential Evolution	0.0
Sphere	Differential Evolution	0.0
Sum Squares	Differential Evolution	0.0
Trid	Genetic Algorithm	0.0
Booth	Genetic algorithm / Particle Swarm Optimization	0.0
Michalewicz	Particle Swarm Optimization	-43.92854152286621
Matyas	Differential evolution	0.0
McCormick	Genetic Algorithm	-23.904371530109593
Cross-in-tray	GA / DE / PSO	-2.0626118708227397
Drop Wave	Genetic algorithm / Particle Swarm Optimization	-1.0
Holder Table	Differential Evolution	-19.208502567886736

Chapter 6

Conclusion

This chapter summarizes the performances of genetic algorithm, differential evolution and particle swarm optimization optimizing the 15 mathematical functions, and bring up several potential research directions based on the results.

6.1 Summary of Experiments

Genetic algorithm performs well in optimizing mathematical function regardless of its existence of local minimum. For instance, Schwefel function has many local minimum, and genetic algorithm get the best solution compared with the differential evolution and particle swarm optimization. Furthermore, genetic algorithm gets the global minimum value for the Booth function, which does not have local minimum.

Differential evolution performs well in optimizing the mathematical functions, as it finds 8 global minimum out of the 15 functions. The longest running time of differential evolution is 54 seconds on optimizing sum of different powers function, which suggests that it is efficient in optimizing the functions.

Compared with genetic algorithm and differential evolution, the experiment performance of particle swarm optimization is not satisfying, as it only finds 2 global minimum out of the 15 mathematical functions. The reason is that more tuning parameters are needed, since the options are not fully tested. The number of particles influences the performance of optimization. For some functions, larger number of particles helps to find the global minimum, while in other cases, smaller number of particles leads to better results.

All of the three algorithms have the attribute of randomness. The randomness lets the algorithms explore the mathematical functions more efficiently and avoid getting trapped in the local minimum, which helps to find the global minimum.

All of the three algorithms are efficient in solving the mathematical functions. All of the experiments took no more than 3 minutes.

6.2 Limitations

From the Table 5.1, there are 10 functions that are optimized with global minimum, and for the other 5 functions, the global minimum have not been found. Some functions such as cross-in-tray function, the local minimum (-2.0626118708227397) is close to the global minimum -2.06261, while some functions are not optimized well. More experiments need to be conducted.

When implementing genetic algorithm, differential evolution and particle swarm optimization, tuning parameters is important. Every parameter may influence the optimization differently, and it is complicated to get the best combination for solving mathematical functions. Proper tuning parameters is needed to improve the result.

6.3 Future Work

To have better performances in optimizing mathematical functions, following directions may be useful:

- Find the proper algorithm to optimize the functions. Different algorithms may have different performances on the same functions. Researchers need to understand the theory and advantages of algorithms, and select the right algorithm.
- Tuning parameters is important in optimizing the mathematical functions, since there are countless combinations of a list of parameters, and it is necessary to find the parameters with good performance.
- Setting suitable stopping criteria is also essential, which determines the result and running time directly. Proper stopping criteria can be acquired by analyzing the experiments and results.
- From the perspective of solving real world issues such as asset allocation, evolutionary algorithms can be used. When solving an issue, we can classify the problem regarding to its attributes, and convert it to a mathematical function. Based on the 15 mathematical functions we have optimized, find the most similar function and apply the best algorithm. After a series of tuning parameters, a proper solution is promising to get.

Bibliography

Albadr, Musatafa Abbas et al. [2020]. "Genetic algorithm based on natural selection theory for optimization problems". In: *Symmetry* 12.11, p. 1758.

Baresel, André, Harmen Sthamer, and Michael Schmidt [2002]. "Fitness function design to improve evolutionary structural testing". In: *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pp. 1329–1336.

Charbonneau, Paul et al. [2002]. "An introduction to genetic algorithms for numerical optimization". In: *NCAR Technical Note* 74, pp. 4–13.

Chudasama, Chetan, SM Shah, and Mahesh Panchal [2011]. "Comparison of parents selection methods of genetic algorithm for TSP". In: *International conference on computer communication and networks CSI-COMNET-2011, Proceedings*. Vol. 85, p. 87.

DEAP 1.4.1 documentation [2023]. Accessed: 2024-08-08. URL: `https://deap.readthedocs.io/en/master/`.

Deb, Kalyanmoy [2000]. "An efficient constraint handling method for genetic algorithms". In: *Computer methods in applied mechanics and engineering* 186.2-4, pp. 311–338.

Deb, Kalyanmoy, Ashish Anand, and Dhiraj Joshi [2002]. "A computationally efficient evolutionary algorithm for real-parameter optimization". In: *Evolutionary computation* 10.4, pp. 371–395.

Eberhart, Russell, and James Kennedy [1995]. "Particle swarm optimization". In: *Proceedings of the IEEE international conference on neural networks*. Vol. 4. Citeseer, pp. 1942–1948.

Gämperle, Roger, Sibylle D Müller, and Petros Koumoutsakos [2002]. "A parameter study for differential evolution". In: *Advances in intelligent systems, fuzzy systems, evolutionary computation* 10.10, pp. 293–298.

Jebari, Khalid, Mohammed Madiafi, et al. [2013]. "Selection methods for genetic algorithms". In: *International Journal of Emerging Sciences* 3.4, pp. 333–344.

Jin, Yaochu, Markus Olhofer, and Bernhard Sendhoff [2001]. "Dynamic weighted aggregation for evolutionary multi-objective optimization: Why does it work

and how". In: *Proceedings of the genetic and evolutionary computation conference*, pp. 1042–1049.

Juneja, Mudita, and SK Nagar [2016]. "Particle swarm optimization algorithm and its parameters: A review". In: *2016 International Conference on Control, Computing, Communication and Materials (ICCCCM)*. IEEE, pp. 1–5.

Katoch, Sourabh, Sumit Singh Chauhan, and Vijay Kumar [2021]. "A review on genetic algorithm: past, present, and future". In: *Multimedia tools and applications* 80, pp. 8091–8126.

Lambora, Annu, Kunal Gupta, and Kriti Chopra [2019]. "Genetic algorithm-A literature review". In: *2019 international conference on machine learning, big data, cloud and parallel computing (COMITCon)*. IEEE, pp. 380–384.

Rocca, Paolo, Giacomo Oliveri, and Andrea Massa [2011]. "Differential evolution as applied to electromagnetics". In: *IEEE Antennas and Propagation Magazine* 53.1, pp. 38–49.

Sedighizadeh, Davoud, and Ellips Masehian [2009]. "Particle swarm optimization methods, taxonomy and applications". In: *International journal of computer theory and engineering* 1.5, pp. 486–502.

Storn, Rainer [1996]. "On the usage of differential evolution for function optimization". In: *Proceedings of North American fuzzy information processing*. Ieee, pp. 519–523.

Van Veldhuizen, David A, and Gary B Lamont [1998]. *Multiobjective evolutionary algorithm research: A history and analysis*. Tech. rep. Citeseer.

Wang, Dongshu, Dapei Tan, and Lei Liu [Jan. 2017]. "Particle swarm optimization algorithm: an overview". In: *Soft Computing* 22.2, 387â408. ISSN: 1433-7479. DOI: 10.1007/s00500-016-2474-6. URL: http://dx.doi.org/10.1007/s00500-016-2474-6.

Yuan, Hongyi et al. [2023]. "A Multi-Channel Frequency Router Based on an Optimization Algorithm and Dispersion Engineering". In: *Nanomaterials* 13.14. ISSN: 2079-4991. DOI: 10.3390/nano13142133. URL: https://www.mdpi.com/2079-4991/13/14/2133.