Evolutionary Computing Algorithm: From Procedural to Object-Oriented Implementation

Ziheng Wang
202283890020
Waterford Institute
Nanjing University of Information Science and Technology
Nanjing, Jiangsu 210044
zhwang@nuist.edu.cn

Abstract

This report studies the transformation of the Cooperative Particle Optimization (CPO) algorithm from procedural programming to object-oriented design. By comparing the original procedural implementation with the object-oriented refactored version, we evaluate the improvements in modularity, maintainability, and extensibility of the algorithm. In the 9 benchmark function tests, the performance of the two implementations is comparable, but the object-oriented approach significantly improves the clarity of the code structure and the convenience of future extensions.

1 Introduction

In this report, we explore the implementation of the Crested Porcupine Optimizer (CPO)[1], a nature-inspired metaheuristic algorithm introduced by Mohamed Abdel-Basset et al. in their paper "Crested Porcupine Optimizer: A New Nature-Inspired Metaheuristic". The primary goal of this report is to demonstrate how object-oriented programming (OOP) principles, specifically the use of classes and objects, can be applied to express an evolutionary computing algorithm such as CPO.

The original procedural implementation of the CPO algorithm directly manipulates particles and fitness functions in a linear fashion. However, as the complexity of the problem increases, this approach can become difficult to manage and extend. Object-oriented programming offers a more effective solution by organizing the algorithm's components into classes. This enhances the modularity, readability, and reusability of the code.

In this report, the CPO algorithm is refactored using OOP concepts, including private and protected members, inheritance, and polymorphism. By utilizing classes to encapsulate the different components of the algorithm, such as particles, fitness functions, and the optimizer itself, we aim to make the code more maintainable and flexible for future improvements. Moreover, the use of OOP techniques allows for better handling of complex features, such as inheritance for different types of particles or fitness functions, and polymorphism to manage various optimization strategies [2].

This report compares the original procedural version of CPO with the modified object-oriented version, focusing on the structural improvements and performance outcomes. Additionally, the effectiveness of the CPO algorithm is validated through a tests .

Embedded Script Development Course Report, Nanjing University of Information Science and Technology (2024).

2 Common Test Functions Used for CPO Evaluation

The following table lists the nine benchmark functions commonly used to evaluate the performance of the CPO algorithm. These functions are diverse and provide various challenges for optimization algorithms.

Table 1: Common Test Functions Used for CPO Evaluation

Name	Function
Sphere	$F_1(x) = \sum_{i=1}^{D} x_i^2$
Schwefel's 2.22	$F_2(x) = \sum_{i=1}^{D} \lvert x_i\rvert + \prod_{i=1}^{D} \lvert x_i\rvert$
Powell Sum	$F_3(x) = \sum_{i=1}^{D} \lvert x_i\rvert^{i+1}$
Schwefel's 1.2	$F_4(x) = \sum_{i=1}^{D} \left(\sum_{j=1}^{i} x_j \right)^2$
Schwefel's 2.21	$F_5(x) = \max\left(\lvert x_i\rvert,\ 1 \le i \le D\right)$
Rosenbrock	$F_6(x) = \sum_{i=1}^{D-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$
Step	$F_7(x) = \sum_{i=1}^{D} (x_i + 0.5)^2$
Quartic	$F_8(x) = \sum_{i=1}^{D} i x_i^4 + \text{random}[0,1]$
Zakharov	$F_9(x) = \sum_{i=1}^{D} x_i^2 + \left(\sum_{i=1}^{D} 0.5 i x_i \right)^2 + \left(\sum_{i=1}^{D} 0.5 i x_i \right)^4$

3 Original Code Overview

The original code, `CPO_algorithm_v4.py`, implements the CPO algorithm using a procedural approach. It defines the nine benchmark functions for optimization and iteratively updates the particles' positions based on global fitness values. The progress of the optimization is monitored through various metrics such as fitness values, convergence curves, and population diversity. Additionally, a 3D visualization is included to display the search history.

However, as the problem becomes more complex and requires additional features such as new functions or visualization improvements, the procedural approach becomes harder to extend and maintain.

4 Modified Code Overview

The modified version of the CPO (Cognitive Particle Optimization) algorithm, implemented in `CPO_algorithm_v4_class.py`, adopts an object-oriented approach. The main improvements include the use of classes to structure the algorithm, and the application of key object-oriented concepts such as private and protected members, inheritance, and polymorphism. This section will walk through the changes made to the code and explain how these concepts were applied.

4.1 Class Structure

The algorithm is encapsulated in the `CPOOptimizer` class, which represents the core of the optimization process. By using classes, we achieve modularity and better organization. The class is responsible for initializing the population, evaluating fitness, and performing optimization steps.

```
class CPOOptimizer:
    def __init__(self, pop_size, Tmax, ub, lb, dim, func_num):
        self.pop_size = pop_size  # Population size
        self.Tmax = Tmax          # Maximum iterations
        self.ub = ub              # Upper bound of search space
        self.lb = lb              # Lower bound of search space
        self.dim = dim            # Dimensionality of the problem
        self.func_num = func_num  # Function number to select the benchmark
        self.benchmark_func = BenchmarkFunction.get_function(func_num)  # Selecting the benchmark function
```

Listing 1: The `CPOOptimizer` class definition

In this class, the constructor (`__init__`) initializes key parameters for the optimization process.

4.2 Private and Protected Members

Private and protected members help ensure that important attributes or methods are not accessed or modified externally. For example, the global best solution (`_Gb_Sol`) and fitness values (`_fitness`) are protected, and the population initialization method (`_initialization()`) is private to restrict its use to within the class.

```
class CPOOptimizer:
    def __init__(self, pop_size, Tmax, ub, lb, dim, func_num):
        self._Gb_Sol = None  # Protected: Global best solution
        self._fitness = None  # Protected: Fitness values for each individual

    def _initialization(self):  # Private method to initialize the population
        return np.random.rand(self.pop_size, self.dim) * (self.ub - self.lb) + self.lb
```

Listing 2: Private and Protected Members Example

The `_initialization` method is used to generate the initial population within the specified bounds.

4.3 Inheritance

Inheritance allows for the creation of subclasses that extend the functionality of the base class. In this implementation, a new class, `CustomCPOOptimizer`, inherits from `CPOOptimizer` and can introduce additional parameters or override methods.

```python
class CustomCPOOptimizer(CPOOptimizer):
    def __init__(self, pop_size, Tmax, ub, lb, dim, func_num,
        new_param):
        super().__init__(pop_size, Tmax, ub, lb, dim, func_num)
        self.new_param = new_param  # Additional parameter for the
            custom optimizer

    def optimize(self):
        # Custom optimization logic, potentially overriding the parent
             method
        pass
```

Listing 3: Inheritance Example

In this case, the `CustomCPOOptimizer` class extends `CPOOptimizer`, adding a new parameter (`new_param`) and potentially modifying the `optimize()` method for specialized optimization strategies.

4.4 Polymorphism

Polymorphism enables methods with the same name to behave differently based on the object type. In the context of this algorithm, the `optimize()` method can be implemented differently in subclasses, allowing for flexible optimization strategies.

```python
class CPOOptimizer:
    def optimize(self):
        # Default optimization behavior
        pass

class AdvancedCPOOptimizer(CPOOptimizer):
    def optimize(self):
        # Custom optimization behavior
        pass

# Usage
optimizer = CPOOptimizer()
optimizer.optimize()  # Calls the base class optimize

advanced_optimizer = AdvancedCPOOptimizer()
advanced_optimizer.optimize()  # Calls the overridden method in
    AdvancedCPOOptimizer
```

Listing 4: Polymorphism Example

Here, both `CPOOptimizer` and `AdvancedCPOOptimizer` implement their own version of the `optimize()` method. Depending on the object type, the appropriate method is called, demonstrating polymorphism in action.

4.5 Benchmark Function Class

The benchmark functions are encapsulated in the `BenchmarkFunction` class, which provides static methods to compute the fitness of particles. This class allows for easy selection of the benchmark function based on a given function number.

```python
class BenchmarkFunction:
    @staticmethod
    def sphere_func(x):
        return np.sum(x ** 2)

    @staticmethod
    def rosenbrock_func(x):
        return np.sum([100 * (x[i + 1] - x[i] ** 2) ** 2 + (x[i] - 1)
            ** 2 for i in range(len(x) - 1)])

```

```python
    @classmethod
    def get_function(cls, func_num):
        functions = {
            1: cls.sphere_func,
            2: cls.rosenbrock_func,
        }
        return functions.get(func_num, None)
```
Listing 5: Benchmark Function Class Example

This class defines two benchmark functions, `sphere_func` and `rosenbrock_func`, and provides a class method (`get_function`) to select a function based on the input parameter `func_num`.

4.6 GitHub Repository

To make the code more accessible and collaborative, we have updated our GitHub repository to include this object-oriented version of the CPO algorithm. The repository now contains the updated code along with comprehensive documentation to help users understand the algorithm and how to extend it. You can access the updated code and additional resources at the following link: https://github.com/Nickory/CPO-python.

5 Experimental Results

The experimental results from both the original and modified codes are presented below, showing the final best fitness values for each of the nine benchmark functions. The results indicate that, overall, the performance of the modified code is very similar to that of the original code, with minor variations attributable to inherent randomness in the optimization process.

The percentage changes between the two versions have been calculated, but the variations observed are negligible and can be attributed to the random nature of the algorithm, especially in terms of initial particle positions and fitness evaluations. Therefore, the core functionality and efficiency of the algorithm remain consistent despite the refactoring from a procedural to an object-oriented approach.

Function	Original Code Final Best Fitness	Modified Code Final Best Fitness
F1	$4.9655733128 \times 10^{-122}$	$5.9551532232 \times 10^{-121}$
F2	$4.5951624409 \times 10^{-120}$	$4.1164518018 \times 10^{-126}$
F3	$2.5603657564 \times 10^{-35}$	$5.6594032075 \times 10^{-36}$
F4	$1.6617729239 \times 10^{-63}$	$3.3723032362 \times 10^{-63}$
F5	$1.1848213366 \times 10^{-115}$	$6.2723264606 \times 10^{-126}$
F6	$9.3614980939 \times 10^{-39}$	$1.0905210557 \times 10^{-2}$
F7	$5.1850501551 \times 10^{-3}$	$3.3886481637 \times 10^{-3}$
F8	1.950648×10^{-17}	$5.6810629015 \times 10^{-18}$
F9	$3.1491281229 \times 10^{-8}$	$7.6923427342 \times 10^{-8}$

Table 2: Comparison of Results Between the Original and Modified Code

6 Conclusion

The comparison between the procedural and object-oriented implementations of the CPO algorithm reveals several key differences in terms of performance and code structure. While the procedural implementation achieves comparable results in terms of fitness values for the test functions, the object-oriented approach offers a more modular and scalable solution. The object-oriented version enables easier future modifications and extensions, which is essential for maintaining the algorithm as it evolves.

The performance differences observed in some benchmark functions can be attributed to the inherent stochastic nature of evolutionary algorithms. Despite these differences, the object-oriented code provides a clearer and more maintainable structure, making it the preferred choice for future developments and experiments.

Adopting object-oriented principles for complex optimization algorithms, such as the CPO algorithm, not only enhances code maintainability but also facilitates the introduction of new features and improvements. This approach aligns well with the evolving nature of computational intelligence techniques and their growing complexity.

7 Code

```python

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from cec2017.functions import all_functions

# Initialization function
def initialization(pop_size, dim, ub, lb):
    return np.random.rand(pop_size, dim) * (ub - lb) + lb


# Benchmark functions from PDF
def sphere_func(x):
    return np.sum(x ** 2)


def schwefel_222_func(x):
    return np.sum(np.abs(x)) + np.prod(np.abs(x))


def powell_sum_func(x):
    return np.sum(np.abs(x) ** (np.arange(len(x)) + 1))


def schwefel_12_func(x):
    return np.sum([np.sum(x[:i + 1]) ** 2 for i in range(len(x))])


def schwefel_221_func(x):
    return np.max(np.abs(x))


def rosenbrock_func(x):
    return np.sum([100 * (x[i + 1] - x[i] ** 2) ** 2 + (x[i] - 1) ** 2
         for i in range(len(x) - 1)])


def step_func(x):
    return np.sum((x + 0.5) ** 2)


def quartic_func(x):
    return np.sum((np.arange(1, len(x) + 1) * x ** 4)) + np.random.
        uniform(0, 1)


def zakharov_func(x):
    term1 = np.sum(x ** 2)
    term2 = np.sum(0.5 * np.arange(1, len(x) + 1) * x) ** 2
    term3 = np.sum(0.5 * np.arange(1, len(x) + 1) * x) ** 4
    return term1 + term2 + term3


# Wrapper for test functions
def fhd(x, func_num):
    if func_num == 1:
        return sphere_func(x)
```

```
    elif func_num == 2:
        return schwefel_222_func(x)
    elif func_num == 3:
        return powell_sum_func(x)
    elif func_num == 4:
        return schwefel_12_func(x)
    elif func_num == 5:
        return schwefel_221_func(x)
    elif func_num == 6:
        return rosenbrock_func(x)
    elif func_num == 7:
        return step_func(x)
    elif func_num == 8:
        return quartic_func(x)
    elif func_num == 9:
        return zakharov_func(x)
    else:
        raise ValueError("Invalid function number")








# CPO main algorithm
def CPO(pop_size, Tmax, ub, lb, dim, func_num):
    Gb_Fit = np.inf
    Gb_Sol = None
    Conv_curve = np.zeros(Tmax)
    X = initialization(pop_size, dim, ub, lb)
    fitness = np.array([fhd(X[i, :], func_num) for i in range(pop_size)])
    Gb_Fit, index = np.min(fitness), np.argmin(fitness)
    Gb_Sol = X[index, :]
    Xp = np.copy(X)
    opt = 0
    t = 0

    while t < Tmax and Gb_Fit > opt:
        for i in range(len(X)):
            U1 = np.random.rand(dim) > np.random.rand(dim)
            rand_index1 = np.random.randint(len(X))
            rand_index2 = np.random.randint(len(X))

            if np.random.rand() < np.random.rand():
                y = (X[i, :] + X[rand_index1, :]) / 2
                X[i, :] = X[i, :] + np.random.randn(dim) * np.abs(2 * np.random.rand() * Gb_Sol - y)
            else:
                Yt = 2 * np.random.rand() * (1 - t / Tmax) ** (t / Tmax)
                U2 = np.random.rand(dim) < 0.5
                S = np.random.rand() * U2
                if np.random.rand() < 0.8:
                    St = np.exp(fitness[i] / (np.sum(fitness) + np.finfo(float).eps))
                    S = S * Yt * St
                    X[i, :] = (1 - U1) * X[i, :] + U1 * (
                                X[rand_index1, :] + St * (X[rand_index2, :] - X[rand_index1, :]) - S)
                else:
```

```python
                    Mt = np.exp(fitness[i] / (np.sum(fitness) + np.finfo(float).eps))
                    Vtp = X[rand_index1, :]
                    Ft = np.random.rand(dim) * (Mt * (-X[i, :] + Vtp))
                    S = S * Yt * Ft
                    X[i, :] = (Gb_Sol + (0.2 * (1 - np.random.rand()) + np.random.rand()) * (U2 * Gb_Sol - X[i, :])) - S

            X[i, :] = np.clip(X[i, :], lb, ub)
            nF = fhd(X[i, :], func_num)
            if fitness[i] < nF:
                X[i, :] = Xp[i, :]
            else:
                Xp[i, :] = X[i, :]
                fitness[i] = nF
                if nF <= Gb_Fit:
                    Gb_Sol = X[i, :]
                    Gb_Fit = nF

        Conv_curve[t] = Gb_Fit
        t += 1
    return Gb_Fit, Gb_Sol, Conv_curve


# Visualize function in 3D
def visualize_function(func, func_num, lb=-10, ub=10, dim=2):
    x = np.linspace(lb, ub, 100)
    y = np.linspace(lb, ub, 100)
    X, Y = np.meshgrid(x, y)
    Z = np.array([func(np.array([x, y]), func_num) for x, y in zip(np.ravel(X), np.ravel(Y))])
    Z = Z.reshape(X.shape)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.plot_surface(X, Y, Z, cmap='viridis')
    ax.set_title(f"Function F{func_num} Visualization")
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("f(x, y)")
    plt.show()


# Run experiments and visualize convergence
def run_experiments(func_num, dim=10, pop_size=50, Tmax=500, ub=100, lb=-100):
    Gb_Fit, Gb_Sol, Conv_curve = CPO(pop_size, Tmax, ub, lb, dim, func_num)

    # Plot convergence curve
    plt.figure()
    plt.plot(Conv_curve, label=f"F{func_num} Convergence")
    plt.xlabel("Iterations")
    plt.ylabel("Best Fitness")
    plt.title(f"Convergence Curve for Function F{func_num}")
    plt.legend()
    plt.show()

    print(f"Function F{func_num} Final Best Fitness: {Gb_Fit:.10e}")


# Main testing loop
for func_num in range(1, 10):
    print(f"Running Function F{func_num}...")
```

```
    visualize_function(fhd, func_num)  # Visualize function in 3D
    run_experiments(func_num)  # Run CPO and plot convergence curve
```

Listing 6: CPO_algorithm_v4.py

```

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


class BenchmarkFunction:
    """
    This class provides various benchmark functions for optimization
        algorithms.
    All functions are static methods, as they do not depend on
        instance attributes.
    """

    @staticmethod
    def sphere_func(x):
        """Sphere function: f(x) = sum(x_i^2)"""
        return np.sum(x ** 2)

    @staticmethod
    def schwefel_222_func(x):
        """Schwefel 2.22 function: f(x) = sum(abs(x_i)) + prod(abs(x_i
            ))"""
        return np.sum(np.abs(x)) + np.prod(np.abs(x))

    @staticmethod
    def powell_sum_func(x):
        """Powell sum function: f(x) = sum(abs(x_i)^(i+1))"""
        return np.sum(np.abs(x) ** (np.arange(len(x)) + 1))

    @staticmethod
    def schwefel_12_func(x):
        """Schwefel 1.2 function: f(x) = sum(sum(x_1...x_i)^2)"""
        return np.sum([np.sum(x[:i + 1]) ** 2 for i in range(len(x))])

    @staticmethod
    def schwefel_221_func(x):
        """Schwefel 2.21 function: f(x) = max(abs(x_i))"""
        return np.max(np.abs(x))

    @staticmethod
    def rosenbrock_func(x):
        """Rosenbrock function: f(x) = sum(100*(x_i+1 - x_i^2)^2 + (
            x_i - 1)^2)"""
        return np.sum([100 * (x[i + 1] - x[i] ** 2) ** 2 + (x[i] - 1)
            ** 2 for i in range(len(x) - 1)])

    @staticmethod
    def step_func(x):
        """Step function: f(x) = sum((x_i + 0.5)^2)"""
        return np.sum((x + 0.5) ** 2)

    @staticmethod
    def quartic_func(x):
        """Quartic function with noise: f(x) = sum(i*x_i^4) + random
            noise"""
        return np.sum((np.arange(1, len(x) + 1) * x ** 4)) + np.random
            .uniform(0, 1)

    @staticmethod
```

```python
    def zakharov_func(x):
        """Zakharov function: f(x) = sum(x_i^2) + sum(0.5*i*x_i)^2 + sum(0.5*i*x_i)^4"""
        term1 = np.sum(x ** 2)
        term2 = np.sum(0.5 * np.arange(1, len(x) + 1) * x) ** 2
        term3 = np.sum(0.5 * np.arange(1, len(x) + 1) * x) ** 4
        return term1 + term2 + term3

    @classmethod
    def get_function(cls, func_num):
        """
        Returns the function corresponding to the func_num.
        """
        functions = {
            1: cls.sphere_func,
            2: cls.schwefel_222_func,
            3: cls.powell_sum_func,
            4: cls.schwefel_12_func,
            5: cls.schwefel_221_func,
            6: cls.rosenbrock_func,
            7: cls.step_func,
            8: cls.quartic_func,
            9: cls.zakharov_func
        }
        if func_num not in functions:
            raise ValueError("Invalid function number")
        return functions[func_num]


class CPOOptimizer:
    """
    CPOOptimizer implements the CPO (Cognitive Particle Optimization) algorithm.
    It includes the population initialization and optimization steps.
    """

    def __init__(self, pop_size, Tmax, ub, lb, dim, func_num):
        """
        Initialize the optimizer with key parameters.

        :param pop_size: Population size
        :param Tmax: Maximum iterations
        :param ub: Upper bound for the search space
        :param lb: Lower bound for the search space
        :param dim: Dimensionality of the problem
        :param func_num: Benchmark function number
        """
        self.pop_size = pop_size
        self.Tmax = Tmax
        self.ub = ub
        self.lb = lb
        self.dim = dim
        self.func_num = func_num
        self.benchmark_func = BenchmarkFunction.get_function(func_num)

    def _initialization(self):
        """Initialize the population with random solutions within bounds."""
        return np.random.rand(self.pop_size, self.dim) * (self.ub - self.lb) + self.lb

    def optimize(self):
        """
        Perform the CPO optimization algorithm.

```

```python
        :return: Best fitness, best solution, and convergence curve
        """
        # Initialize variables
        Gb_Fit = np.inf
        Gb_Sol = None
        Conv_curve = np.zeros(self.Tmax)
        X = self._initialization()
        fitness = np.array([self.benchmark_func(X[i, :]) for i in range(self.pop_size)])
        Gb_Fit, index = np.min(fitness), np.argmin(fitness)
        Gb_Sol = X[index, :]
        Xp = np.copy(X)
        opt = 0
        t = 0

        # Main optimization loop
        while t < self.Tmax and Gb_Fit > opt:
            for i in range(len(X)):
                U1 = np.random.rand(self.dim) > np.random.rand(self.dim)
                rand_index1 = np.random.randint(len(X))
                rand_index2 = np.random.randint(len(X))

                if np.random.rand() < np.random.rand():
                    y = (X[i, :] + X[rand_index1, :]) / 2
                    X[i, :] = X[i, :] + np.random.randn(self.dim) * np.abs(2 * np.random.rand() * Gb_Sol - y)
                else:
                    Yt = 2 * np.random.rand() * (1 - t / self.Tmax) ** (t / self.Tmax)
                    U2 = np.random.rand(self.dim) < 0.5
                    S = np.random.rand() * U2
                    if np.random.rand() < 0.8:
                        St = np.exp(fitness[i] / (np.sum(fitness) + np.finfo(float).eps))
                        S = S * Yt * St
                        X[i, :] = (1 - U1) * X[i, :] + U1 * (
                                X[rand_index1, :] + St * (X[rand_index2, :] - X[rand_index1, :]) - S)
                    else:
                        Mt = np.exp(fitness[i] / (np.sum(fitness) + np.finfo(float).eps))
                        Vtp = X[rand_index1, :]
                        Ft = np.random.rand(self.dim) * (Mt * (-X[i, :] + Vtp))
                        S = S * Yt * Ft
                        X[i, :] = (Gb_Sol + (0.2 * (1 - np.random.rand()) + np.random.rand()) * (
                                U2 * Gb_Sol - X[i, :])) - S

                X[i, :] = np.clip(X[i, :], self.lb, self.ub)
                nF = self.benchmark_func(X[i, :])
                if fitness[i] < nF:
                    X[i, :] = Xp[i, :]
                else:
                    Xp[i, :] = X[i, :]
                    fitness[i] = nF
                    if nF <= Gb_Fit:
                        Gb_Sol = X[i, :]
                        Gb_Fit = nF

            Conv_curve[t] = Gb_Fit
            t += 1

```

```
        return Gb_Fit, Gb_Sol, Conv_curve


class Visualization:
    """
    The Visualization class handles the plotting and visualization of optimization results.
    """

    @staticmethod
    def visualize_function(benchmark_func, func_num, lb=-10, ub=10, dim=2):
        """
        Visualize the benchmark function in 3D.

        :param benchmark_func: Function to visualize
        :param func_num: Function number
        :param lb: Lower bound for the axes
        :param ub: Upper bound for the axes
        :param dim: Dimensionality of the problem
        """
        x = np.linspace(lb, ub, 100)
        y = np.linspace(lb, ub, 100)
        X, Y = np.meshgrid(x, y)
        Z = np.array([benchmark_func(np.array([x, y])) for x, y in zip(np.ravel(X), np.ravel(Y))])
        Z = Z.reshape(X.shape)

        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        ax.plot_surface(X, Y, Z, cmap='viridis')
        ax.set_title(f"Function F{func_num} Visualization")
        ax.set_xlabel("x")
        ax.set_ylabel("y")
        ax.set_zlabel("f(x, y)")
        plt.show()

    @staticmethod
    def plot_convergence(Conv_curve, func_num):
        """
        Plot the convergence curve of the optimization.

        :param Conv_curve: The convergence curve data
        :param func_num: Function number
        """
        plt.figure()
        plt.plot(Conv_curve, label=f"F{func_num} Convergence")
        plt.xlabel("Iterations")
        plt.ylabel("Best Fitness")
        plt.title(f"Convergence Curve for Function F{func_num}")
        plt.legend()
        plt.show()


def run_experiments(func_num, dim=10, pop_size=50, Tmax=500, ub=100, lb=-100):
    """
    Run optimization experiments for a specific benchmark function.

    :param func_num: Function number
    :param dim: Dimensionality of the problem
    :param pop_size: Population size
    :param Tmax: Maximum iterations
    :param ub: Upper bound for the search space
    :param lb: Lower bound for the search space
```

```python
    """
    benchmark_func = BenchmarkFunction.get_function(func_num)
    optimizer = CPOOptimizer(pop_size, Tmax, ub, lb, dim, func_num)
    Gb_Fit, Gb_Sol, Conv_curve = optimizer.optimize()

    # Visualizations
    Visualization.visualize_function(benchmark_func, func_num)
    Visualization.plot_convergence(Conv_curve, func_num)

    print(f"Function F{func_num} Final Best Fitness: {Gb_Fit:.10e}")


def main():
    """
    Main function to run the experiments for all benchmark functions.
    """
    for func_num in range(1, 10):
        print(f"Running Function F{func_num}...")
        run_experiments(func_num)


if __name__ == "__main__":
    main()
```

Listing 7: CPO_algorithm_v4_class.py

References

[1] Mohamed Abdel-Basset, Reda Mohamed, and Mohamed Abouhawwash. Crested porcupine optimizer: A new nature-inspired metaheuristic. *Knowledge-Based Systems*, 284:111257, 2024.

[2] Tim Rentsch. Object oriented programming. *ACM Sigplan Notices*, 17(9):51–57, 1982.