

---

# 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 .

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

## 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 `CP0_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 `CP0Optimizer` 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.

```

1 class CP0Optimizer:
2     def __init__(self, pop_size, Tmax, ub, lb, dim, func_num):
3         self.pop_size = pop_size      # Population size
4         self.Tmax = Tmax               # Maximum iterations
5         self.ub = ub                  # Upper bound of search space
6         self.lb = lb                  # Lower bound of search space
7         self.dim = dim                # Dimensionality of the problem
8         self.func_num = func_num      # Function number to select the
          benchmark
9         self.benchmark_func = BenchmarkFunction.get_function(func_num)
          # Selecting the benchmark function

```

Listing 1: The `CP0Optimizer` 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.

```

1 class CP0Optimizer:
2     def __init__(self, pop_size, Tmax, ub, lb, dim, func_num):
3         self._Gb_Sol = None          # Protected: Global best solution
4         self._fitness = None         # Protected: Fitness values for each
          individual
5
6     def _initialization(self):        # Private method to initialize the
          population
7         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, `CustomCP0Optimizer`, inherits from `CP0Optimizer` and can introduce additional parameters or override methods.

```

1 class CustomCP00Optimizer(CP00Optimizer):
2     def __init__(self, pop_size, Tmax, ub, lb, dim, func_num,
3         new_param):
4         super().__init__(pop_size, Tmax, ub, lb, dim, func_num)
5         self.new_param = new_param # Additional parameter for the
6                                     custom optimizer
7
8     def optimize(self):
9         # Custom optimization logic, potentially overriding the parent
10            method
11        pass

```

Listing 3: Inheritance Example

In this case, the CustomCP00Optimizer class extends CP00Optimizer, 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.

```

1 class CP00Optimizer:
2     def optimize(self):
3         # Default optimization behavior
4         pass
5
6 class AdvancedCP00Optimizer(CP00Optimizer):
7     def optimize(self):
8         # Custom optimization behavior
9         pass
10
11 # Usage
12 optimizer = CP00Optimizer()
13 optimizer.optimize() # Calls the base class optimize
14
15 advanced_optimizer = AdvancedCP00Optimizer()
16 advanced_optimizer.optimize() # Calls the overridden method in
17     AdvancedCP00Optimizer

```

Listing 4: Polymorphism Example

Here, both CP00Optimizer and AdvancedCP00Optimizer 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.

```

1 class BenchmarkFunction:
2     @staticmethod
3     def sphere_func(x):
4         return np.sum(x ** 2)
5
6     @staticmethod
7     def rosenbrock_func(x):
8         return np.sum([100 * (x[i + 1] - x[i] ** 2) ** 2 + (x[i] - 1)
9             ** 2 for i in range(len(x) - 1)])

```

```

10     @classmethod
11     def get_function(cls, func_num):
12         functions = {
13             1: cls.sphere_func,
14             2: cls.rosenbrock_func,
15         }
16         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	$2.0604446439 \times 10^{-28}$	$5.0355069046 \times 10^{-29}$
F2	$1.6110891524 \times 10^{-14}$	$1.1659656157 \times 10^{-14}$
F3	$6.8096921314 \times 10^{-31}$	$4.6332349821 \times 10^{-34}$
F4	$2.0760676143 \times 10^{-16}$	$1.3920431973 \times 10^{-18}$
F5	$2.5258077110 \times 10^{-11}$	$2.5453754461 \times 10^{-13}$
F6	$5.8162053778 \times 10^0$	$5.8163331126 \times 10^0$
F7	$7.4122871982 \times 10^{-5}$	$9.6303091013 \times 10^{-5}$
F8	$7.1892964882 \times 10^{-3}$	$2.4062497156 \times 10^{-4}$
F9	$2.4399936896 \times 10^{-9}$	$1.5213113585 \times 10^{-11}$

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

```

1
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from mpl_toolkits.mplot3d import Axes3D
5 from cec2017.functions import all_functions
6
7 # Initialization function
8 def initialization(pop_size, dim, ub, lb):
9     return np.random.rand(pop_size, dim) * (ub - lb) + lb
10
11
12 # Benchmark functions from PDF
13 def sphere_func(x):
14     return np.sum(x ** 2)
15
16
17 def schwefel_222_func(x):
18     return np.sum(np.abs(x)) + np.prod(np.abs(x))
19
20
21 def powell_sum_func(x):
22     return np.sum(np.abs(x) ** (np.arange(len(x)) + 1))
23
24
25 def schwefel_12_func(x):
26     return np.sum([np.sum(x[:i + 1]) ** 2 for i in range(len(x))])
27
28
29 def schwefel_221_func(x):
30     return np.max(np.abs(x))
31
32
33 def rosenbrock_func(x):
34     return np.sum([100 * (x[i + 1] - x[i] ** 2) ** 2 + (x[i] - 1) ** 2
35                     for i in range(len(x) - 1)])
36
37
38 def step_func(x):
39     return np.sum((x + 0.5) ** 2)
40
41
42 def quartic_func(x):
43     return np.sum((np.arange(1, len(x) + 1) * x ** 4)) + np.random.
44         uniform(0, 1)
45
46
47 def zakharov_func(x):
48     term1 = np.sum(x ** 2)
49     term2 = np.sum(0.5 * np.arange(1, len(x) + 1) * x) ** 2
50     term3 = np.sum(0.5 * np.arange(1, len(x) + 1) * x) ** 4
51     return term1 + term2 + term3
52
53 # Wrapper for test functions
54 def fhd(x, func_num):
55     if func_num == 1:
56         return sphere_func(x)

```

```

56     elif func_num == 2:
57         return schwefel_222_func(x)
58     elif func_num == 3:
59         return powell_sum_func(x)
60     elif func_num == 4:
61         return schwefel_12_func(x)
62     elif func_num == 5:
63         return schwefel_221_func(x)
64     elif func_num == 6:
65         return rosenbrock_func(x)
66     elif func_num == 7:
67         return step_func(x)
68     elif func_num == 8:
69         return quartic_func(x)
70     elif func_num == 9:
71         return zakharov_func(x)
72     else:
73         raise ValueError("Invalid function number")
74
75
76
77
78
79
80
81
82 # CPO main algorithm
83 def CPO(pop_size, Tmax, ub, lb, dim, func_num):
84     Gb_Fit = np.inf
85     Gb_Sol = None
86     Conv_curve = np.zeros(Tmax)
87     X = initialization(pop_size, dim, ub, lb)
88     fitness = np.array([fhd(X[i, :], func_num) for i in range(pop_size)
89 ])
89     Gb_Fit, index = np.min(fitness), np.argmin(fitness)
90     Gb_Sol = X[index, :]
91     Xp = np.copy(X)
92     opt = 0
93     t = 0
94
95     while t < Tmax and Gb_Fit > opt:
96         for i in range(len(X)):
97             U1 = np.random.rand(dim) > np.random.rand(dim)
98             rand_index1 = np.random.randint(len(X))
99             rand_index2 = np.random.randint(len(X))
100
101             if np.random.rand() < np.random.rand():
102                 y = (X[i, :] + X[rand_index1, :]) / 2
103                 X[i, :] = X[i, :] + np.random.randn(dim) * np.abs(2 *
104                     np.random.rand() * Gb_Sol - y)
105             else:
106                 Yt = 2 * np.random.rand() * (1 - t / Tmax) ** (t /
107                     Tmax)
108                 U2 = np.random.rand(dim) < 0.5
109                 S = np.random.rand() * U2
110                 if np.random.rand() < 0.8:
111                     St = np.exp(fitness[i] / (np.sum(fitness) + np.
112                         finfo(float).eps))
113                     S = S * Yt * St
114                     X[i, :] = (1 - U1) * X[i, :] + U1 * (
115                         X[rand_index1, :] + St * (X[
116                             rand_index2, :] - X[rand_index1,
117                             :]) - S)
118             else:

```

```

114         Mt = np.exp(fitness[i] / (np.sum(fitness) + np.
115             finfo(float).eps))
116         Vtp = X[rand_index1, :]
117         Ft = np.random.rand(dim) * (Mt * (-X[i, :] + Vtp))
118         S = S * Yt * Ft
119         X[i, :] = (Gb_Sol + (0.2 * (1 - np.random.rand())
120             + np.random.rand()) * (U2 * Gb_Sol - X[i, :]))
121             - S
122
123         X[i, :] = np.clip(X[i, :], lb, ub)
124         nF = fhd(X[i, :], func_num)
125         if fitness[i] < nF:
126             X[i, :] = Xp[i, :]
127         else:
128             Xp[i, :] = X[i, :]
129             fitness[i] = nF
130             if nF <= Gb_Fit:
131                 Gb_Sol = X[i, :]
132                 Gb_Fit = nF
133
134         Conv_curve[t] = Gb_Fit
135         t += 1
136         return Gb_Fit, Gb_Sol, Conv_curve
137
138 # Visualize function in 3D
139 def visualize_function(func, func_num, lb=-10, ub=10, dim=2):
140     x = np.linspace(lb, ub, 100)
141     y = np.linspace(lb, ub, 100)
142     X, Y = np.meshgrid(x, y)
143     Z = np.array([func(np.array([x, y])), func_num] for x, y in zip(np.
144         ravel(X), np.ravel(Y)))
145     Z = Z.reshape(X.shape)
146
147     fig = plt.figure()
148     ax = fig.add_subplot(111, projection='3d')
149     ax.plot_surface(X, Y, Z, cmap='viridis')
150     ax.set_title(f"Function F{func_num} Visualization")
151     ax.set_xlabel("x")
152     ax.set_ylabel("y")
153     ax.set_zlabel("f(x, y)")
154     plt.show()
155
156 # Run experiments and visualize convergence
157 def run_experiments(func_num, dim=10, pop_size=50, Tmax=500, ub=100,
158     lb=-100):
159     Gb_Fit, Gb_Sol, Conv_curve = CP0(pop_size, Tmax, ub, lb, dim,
160         func_num)
161
162     # Plot convergence curve
163     plt.figure()
164     plt.plot(Conv_curve, label=f"F{func_num} Convergence")
165     plt.xlabel("Iterations")
166     plt.ylabel("Best Fitness")
167     plt.title(f"Convergence Curve for Function F{func_num}")
168     plt.legend()
169     plt.show()
170
171     print(f"Function F{func_num} Final Best Fitness: {Gb_Fit:.10e}")
172
173 # Main testing loop
174 for func_num in range(1, 10):
175     print(f"Running Function F{func_num}...")

```



```

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

```

Listing 6: CPO\_algorithm\_v4.py

```

1
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from mpl_toolkits.mplot3d import Axes3D
5
6
7 class BenchmarkFunction:
8     """
9     This class provides various benchmark functions for optimization
10    algorithms.
11    All functions are static methods, as they do not depend on
12    instance attributes.
13    """
14
15    @staticmethod
16    def sphere_func(x):
17        """Sphere function:  $f(x) = \sum(x_i^2)$ """
18        return np.sum(x ** 2)
19
20    @staticmethod
21    def schwefel_222_func(x):
22        """Schwefel 2.22 function:  $f(x) = \sum(\text{abs}(x_i)) + \text{prod}(\text{abs}(x_i))$ """
23        return np.sum(np.abs(x)) + np.prod(np.abs(x))
24
25    @staticmethod
26    def powell_sum_func(x):
27        """Powell sum function:  $f(x) = \sum(\text{abs}(x_i)^{(i+1)})$ """
28        return np.sum(np.abs(x) ** (np.arange(len(x)) + 1))
29
30    @staticmethod
31    def schwefel_12_func(x):
32        """Schwefel 1.2 function:  $f(x) = \sum(\sum(x_1 \dots x_i)^2)$ """
33        return np.sum([np.sum(x[:i + 1]) ** 2 for i in range(len(x))])
34
35    @staticmethod
36    def schwefel_221_func(x):
37        """Schwefel 2.21 function:  $f(x) = \max(\text{abs}(x_i))$ """
38        return np.max(np.abs(x))
39
40    @staticmethod
41    def rosenbrock_func(x):
42        """Rosenbrock function:  $f(x) = \sum(100*(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$ """
43        return np.sum([100 * (x[i + 1] - x[i] ** 2) ** 2 + (x[i] - 1)
44                        ** 2 for i in range(len(x) - 1)])
45
46    @staticmethod
47    def step_func(x):
48        """Step function:  $f(x) = \sum((x_i + 0.5)^2)$ """
49        return np.sum((x + 0.5) ** 2)
50
51    @staticmethod
52    def quartic_func(x):
53        """Quartic function with noise:  $f(x) = \sum(i*x_i^4) + \text{random noise}$ """
54        return np.sum((np.arange(1, len(x) + 1) * x ** 4)) + np.random
55        .uniform(0, 1)
56
57    @staticmethod

```

```

54 def zakharov_func(x):
55     """Zakharov function:  $f(x) = \sum(x_i^2) + \sum(0.5*i*x_i)^2 + \sum(0.5*i*x_i)^4$ """
56     term1 = np.sum(x ** 2)
57     term2 = np.sum(0.5 * np.arange(1, len(x) + 1) * x) ** 2
58     term3 = np.sum(0.5 * np.arange(1, len(x) + 1) * x) ** 4
59     return term1 + term2 + term3
60
61 @classmethod
62 def get_function(cls, func_num):
63     """
64     Returns the function corresponding to the func_num.
65     """
66     functions = {
67         1: cls.sphere_func,
68         2: cls.schwefel_222_func,
69         3: cls.powell_sum_func,
70         4: cls.schwefel_12_func,
71         5: cls.schwefel_221_func,
72         6: cls.rosenbrock_func,
73         7: cls.step_func,
74         8: cls.quartic_func,
75         9: cls.zakharov_func
76     }
77     if func_num not in functions:
78         raise ValueError("Invalid function number")
79     return functions[func_num]
80
81
82 class CP0Optimizer:
83     """
84     CP0Optimizer implements the CP0 (Cognitive Particle Optimization)
85     algorithm.
86     It includes the population initialization and optimization steps.
87     """
88     def __init__(self, pop_size, Tmax, ub, lb, dim, func_num):
89         """
90         Initialize the optimizer with key parameters.
91
92         :param pop_size: Population size
93         :param Tmax: Maximum iterations
94         :param ub: Upper bound for the search space
95         :param lb: Lower bound for the search space
96         :param dim: Dimensionality of the problem
97         :param func_num: Benchmark function number
98         """
99         self.pop_size = pop_size
100        self.Tmax = Tmax
101        self.ub = ub
102        self.lb = lb
103        self.dim = dim
104        self.func_num = func_num
105        self.benchmark_func = BenchmarkFunction.get_function(func_num)
106
107     def _initialization(self):
108         """Initialize the population with random solutions within
109         bounds."""
110
111         return np.random.rand(self.pop_size, self.dim) * (self.ub -
112             self.lb) + self.lb
113
114     def optimize(self):
115         """
116         Perform the CP0 optimization algorithm.

```

```

115         :return: Best fitness, best solution, and convergence curve
116         """
117         # Initialize variables
118         Gb_Fit = np.inf
119         Gb_Sol = None
120         Conv_curve = np.zeros(self.Tmax)
121         X = self._initialization()
122         fitness = np.array([self.benchmark_func(X[i, :]) for i in
123                             range(self.pop_size)])
124         Gb_Fit, index = np.min(fitness), np.argmin(fitness)
125         Gb_Sol = X[index, :]
126         Xp = np.copy(X)
127         opt = 0
128         t = 0
129
130         # Main optimization loop
131         while t < self.Tmax and Gb_Fit > opt:
132             for i in range(len(X)):
133                 U1 = np.random.rand(self.dim) > np.random.rand(self.
134                     dim)
135                 rand_index1 = np.random.randint(len(X))
136                 rand_index2 = np.random.randint(len(X))
137
138                 if np.random.rand() < np.random.rand():
139                     y = (X[i, :] + X[rand_index1, :]) / 2
140                     X[i, :] = X[i, :] + np.random.randn(self.dim) * np
141                         .abs(2 * np.random.rand() * Gb_Sol - y)
142
143                 else:
144                     Yt = 2 * np.random.rand() * (1 - t / self.Tmax) **
145                         (t / self.Tmax)
146                     U2 = np.random.rand(self.dim) < 0.5
147                     S = np.random.rand() * U2
148                     if np.random.rand() < 0.8:
149                         St = np.exp(fitness[i] / (np.sum(fitness) + np
150                             .finfo(float).eps))
151                         S = S * Yt * St
152                         X[i, :] = (1 - U1) * X[i, :] + U1 * (
153                             X[rand_index1, :] + St * (X[
154                                 rand_index2, :] - X[rand_index1,
155                                 :]) - S)
156                     else:
157                         Mt = np.exp(fitness[i] / (np.sum(fitness) + np
158                             .finfo(float).eps))
159                         Vtp = X[rand_index1, :]
160                         Ft = np.random.rand(self.dim) * (Mt * (-X[i,
161                             :] + Vtp))
162                         S = S * Yt * Ft
163                         X[i, :] = (Gb_Sol + (0.2 * (1 - np.random.rand
164                             ())) + np.random.rand()) * (
165                             U2 * Gb_Sol - X[i, :])) - S
166
167                 X[i, :] = np.clip(X[i, :], self.lb, self.ub)
168                 nF = self.benchmark_func(X[i, :])
169                 if fitness[i] < nF:
170                     X[i, :] = Xp[i, :]
171                 else:
172                     Xp[i, :] = X[i, :]
173                     fitness[i] = nF
174                     if nF <= Gb_Fit:
175                         Gb_Sol = X[i, :]
176                         Gb_Fit = nF
177
178             Conv_curve[t] = Gb_Fit
179             t += 1

```

```

170         return Gb_Fit, Gb_Sol, Conv_curve
171
172
173 class Visualization:
174     """
175     The Visualization class handles the plotting and visualization of
176     optimization results.
177     """
178
179     @staticmethod
180     def visualize_function(benchmark_func, func_num, lb=-10, ub=10,
181                           dim=2):
182         """
183         Visualize the benchmark function in 3D.
184
185         :param benchmark_func: Function to visualize
186         :param func_num: Function number
187         :param lb: Lower bound for the axes
188         :param ub: Upper bound for the axes
189         :param dim: Dimensionality of the problem
190         """
191         x = np.linspace(lb, ub, 100)
192         y = np.linspace(lb, ub, 100)
193         X, Y = np.meshgrid(x, y)
194         Z = np.array([benchmark_func(np.array([x, y])) for x, y in zip
195                       (np.ravel(X), np.ravel(Y))])
196         Z = Z.reshape(X.shape)
197
198         fig = plt.figure()
199         ax = fig.add_subplot(111, projection='3d')
200         ax.plot_surface(X, Y, Z, cmap='viridis')
201         ax.set_title(f"Function F{func_num} Visualization")
202         ax.set_xlabel("x")
203         ax.set_ylabel("y")
204         ax.set_zlabel("f(x, y)")
205         plt.show()
206
207     @staticmethod
208     def plot_convergence(Conv_curve, func_num):
209         """
210         Plot the convergence curve of the optimization.
211
212         :param Conv_curve: The convergence curve data
213         :param func_num: Function number
214         """
215         plt.figure()
216         plt.plot(Conv_curve, label=f"F{func_num} Convergence")
217         plt.xlabel("Iterations")
218         plt.ylabel("Best Fitness")
219         plt.title(f"Convergence Curve for Function F{func_num}")
220         plt.legend()
221         plt.show()
222
223 def run_experiments(func_num, dim=10, pop_size=50, Tmax=500, ub=100,
224                    lb=-100):
225     """
226     Run optimization experiments for a specific benchmark function.
227
228     :param func_num: Function number
229     :param dim: Dimensionality of the problem
230     :param pop_size: Population size
231     :param Tmax: Maximum iterations
232     :param ub: Upper bound for the search space
233     :param lb: Lower bound for the search space

```

```

231     """
232     benchmark_func = BenchmarkFunction.get_function(func_num)
233     optimizer = CPOOptimizer(pop_size, Tmax, ub, lb, dim, func_num)
234     Gb_Fit, Gb_Sol, Conv_curve = optimizer.optimize()
235
236     # Visualizations
237     Visualization.visualize_function(benchmark_func, func_num)
238     Visualization.plot_convergence(Conv_curve, func_num)
239
240     print(f"Function F{func_num} Final Best Fitness: {Gb_Fit:.10e}")
241
242
243 def main():
244     """
245     Main function to run the experiments for all benchmark functions.
246     """
247     for func_num in range(1, 10):
248         print(f"Running Function F{func_num}...")
249         run_experiments(func_num)
250
251
252 if __name__ == "__main__":
253     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.