Design and Implementation of a New Lightweight Optimization Algorithm

Summary:

Based on the research on particle swarm optimization algorithm, I propose a new optimization algorithm called clustering algorithm. Compared to particle swarm optimization, swarm optimization has the "lightweight" characteristics of simple principle and small memory consumption, and has shown its stability and speed in handling some problems. This article will introduce its principles, development process, and characteristics demonstrated in the handling of certain optimization problems.

1.1

Particle Swarm Optimization (PSO) is a widely used optimization algorithm in engineering, which introduces the concept of particle swarm and optimizes the objective function by mimicking bird hunting behavior.

Taking the search for the minimum value as an example, each particle movement method is related to two factors, the population best point and the individual historical best point. By randomizing the weights of the two, the optimal solution can be quickly and efficiently found.

And this algorithm also has obvious shortcomings in practical applications. Due to the introduction of particle inertia, the optimal individual in the final particle swarm will oscillate back and forth, resulting in errors that fluctuate greatly; And the algorithm involves numerous parameters of particles, which prolongs the computation time and puts higher demands on memory capacity, while the development process is relatively more complex. For this, I attempted to explore more efficient optimization algorithms based on particle swarm optimization.

Firstly, it should be clarified that the idea behind my proposal of the new algorithm is based on fully absorbing the advantages of particle swarm optimization, that is, the new algorithm is also designed as a group optimization algorithm, which optimizes the objective function through the movement of the group. And in order to make the algorithm as simple and efficient as possible, the design of the new algorithm should involve fewer parameters in principle.

I will name the new algorithm "Crowding Algorithm", and the principle is roughly as follows: First, initialize a group, search for the optimal individual in the group, and then all individuals approach the optimal individual, denoted as $\{\overrightarrow{r_i}\}$ $(1 \leq i \leq n)$. The optimal individual is $_{rm}$, where m $\in[1, n]$ $\forall j \ s.t. \ 1 \leq j \leq n \ and \ j \neq m$ performs the following operations: $\overrightarrow{r_j}' = \overrightarrow{r_j} +$ ra* $(\overrightarrow{r_m} - \overrightarrow{r_j})$, Among them$\in [0.5, 2]$, ra is a random number, and the geometric meaning of this process is

to move all individuals in the group towards the optimal individual in the group, but the movement distance is random, ranging from 0.5 to 2 times the distance from the original individual to the optimal solution.

During the optimization process, the group will converge while moving towards the optimal solution. Moving towards the optimal solution is because each point is moving towards the current group optimal solution, while convergence is due to the fact that the distance from each point to the group optimal solution decreases after moving.

After clarifying the algorithm principle and considering various factors, I chose Python programming language to implement it. Firstly, it is necessary to design objects for individuals within a group. Compared to the individual groups in particle swarm optimization, individuals in clustering algorithms do not need to save the optimal solutions they have reached. Therefore, only a vector class needs to be designed to describe individual masses. Considering the generality of dimensionality, I use a list to initialize the individual vector of this group. The length of the list is the dimension of the vector and also represents the number of parameters to be optimized in the objective function. To facilitate its display, set a show attribute and return this list for easy viewing. After the framework of the object is set up, some methods need to be designed for it, including linear operations such as addition and subtraction, which are performed between this class, as well as multiplication, which is used for operations between this type of data and floating-point and integer data. After completing the basic attribute and method design, additional methods and attributes can be developed to improve the object, such as defining its module length and number of movements, which facilitates research, maintenance, and improvement.

Secondly, it is necessary to design a reasonable objective function. In order to enhance the adaptability of the entire program framework, it is necessary to avoid directly using each parameter as an independent variable in the design of the objective function. Otherwise, once the number of parameters changes, the code for solving the objective function value also needs to be changed. To achieve this, set the independent variables of the objective function to a list. Simply input the list corresponding to the show attribute of the vector to output the objective function value at that point

```
def f(list_l):
    return
math.log((list_l[0]-60)*(list_l[0]-60)+(list_l[1]-80)*(list_l[1]-80)+(list_l[2]-100)*(list_l[2]-100)+(list_l[3]-120)*(list_l[3]-120)+(list_l[4]-160)*(list_l[4]-160)+10)
```

In order to further facilitate the calculation of function values, I added the goal method to the group individual objects, where the independent variable of the method is a function, that is, a stack structured data. And this function as the independent variable is exactly the function mentioned earlier with the list as the independent variable. In this way, when performing this method on individual vector objects in a group, the function value corresponding to the show attribute list of the vector will be returned, which is the objective function value at that point. The final class design is as follows:

```
class vec:
```

```
def __init__(self,l):
    self.show=l
def mul(self,n):
    sl=[]
    for i in range(len(self.show)):
        sl.append(self.show[i]*n)
    return vec(sl)
def __add__(self,other):
    return vec(s(self.show,other.show,1))
def __sub__(self,other):
    return vec(s(self.show,other.show,-1))
def goal(self,f):
    return f(self.show)
def ran(self,m):
    for i in range(len(self.show)):
        self.show[i]+=(m.show[i]-self.show[i])*randint(5000,30000)/10000
```

Finally, regarding the objects of the entire group, since the group itself does not have many operations, as most operations have already been performed on individual members of the group, only one list is needed to load the group itself. Then we need to design a function that seeks optimization in such a list.

In fact, this is a traversal tool that uses the "arena" algorithm to determine the individual in the group that corresponds to the maximum or minimum function value, and returns that individual. After encapsulating these function classes, the main program becomes very simple. If the group name is l and the group individuals are named vec, the first step is to initialize an l and import randint (a random integer function) to start the for loop iteration:

```
for i in range (n):
    lmin=mn(f,l)
    for d in l:
        d+=(lmin-d).mul(randint(5000,20000)/10000)
```

Among them, mn is the function that searches for the minimum value in the group, corresponding to the individual. The program searches for the minimum value, and then uses mul to calculate the difference to complete the movement of group individuals, nested in the traversal of the group.

After implementing the program framework, I designed the following functions:

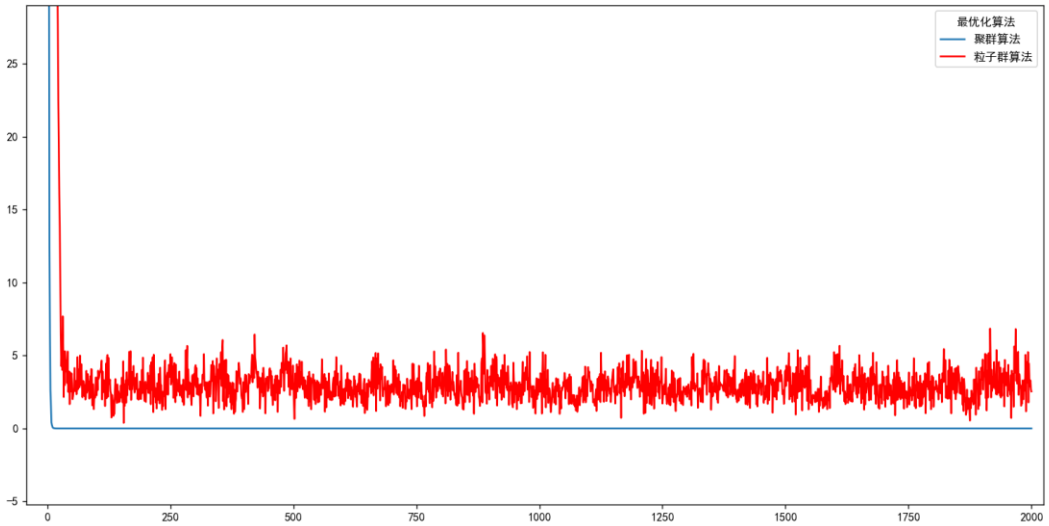
$f(x,y,z,u,v)=\ln[(x-60)^2+(y-80)^2+(z-100)^2+(u-120)^2+(v-160)^2+10]$

It is easy to obtain its minimum value, which is obtained at (60, 80, 100, 120, 160).

Encapsulate this as a function with a list of variables, using l [0] instead of x, l [1] instead of y... until l [4] replaces v, for a total of five variables.

In the test, I compared it with particle swarm optimization algorithm, using the same objective function (as introduced in section 2.1), with the same initial position of the group and the same number of iterations. Given that the selected objective function can theoretically determine the optimal solution, the Euclidean distance between the optimal individual and the optimal solution is calculated after each iteration. Both algorithms not only output their respective results, but also

demonstrate the convergence process. Visualization is based on the matplotlib. piplot library, which serves as a framework to visually demonstrate the convergence process of the two algorithms.



The horizontal axis represents the number of iterations, and the vertical axis represents the Euclidean distance to the optimal solution. It is not difficult to see from the comparison chart that the particle swarm algorithm has added obstacles to the stability of the final result due to the particle attribute of "velocity". It can be seen that the Euclidean distance from the iteration result of particle swarm algorithm to the optimal solution exhibits significant fluctuations, while the clustering algorithm does not have this problem.

This is the principle of the first generation clustering algorithm that I designed.

2.2 Introduction of Inertia Factor

As research progresses, I have found that clustering algorithms have an important drawback. When the optimal solution is not within the maximum range enclosed by the initialization group and is far from the initial group state, the output after multiple iterations is stable, but still far from the optimal solution.

This is a drawback that particle swarm optimization algorithm has not encountered before. Through further research, it is not difficult to theoretically deduce the reason for this phenomenon. For each movement, if the optimal solution in the group is used as a reference, the relative positions of the remaining group individuals are equivalent to multiplying the original basis by a coefficient, and this coefficient is$\in [-1,0.5]$. From this, it can be seen that after each iteration, the group shrinks relative to the optimal individual in the current group. Therefore, in terms of the movement of the group, it is a process of moving towards the optimal solution on one side and gathering towards the optimal individual on the other side.

This is the origin of the name "cluster", but in such cases, it becomes an obstacle to the optimization process. To address this issue, I developed the second-generation clustering algorithm as a transitional version specifically optimized for this problem.

By adding attributes and other tasks to vec objects, I was able to understand the amount of particle motion each time. In the next exercise, it is sufficient to consider the amount of exercise simultaneously. This amount of motion is called 'inertia', which is then multiplied by a constant and a random number between 0.5 and 1 to form the inertia factor. In addition, after this

improvement, the movement of individuals in the group may not necessarily occur on the line connecting them to the optimal individual, but rather on each axis moving independently.

```
 def change(self):
            return vec(s(self.show,self.last,-1))
    def update(self):
            self.last=self.show
    def ran(self,m):
          for i in range(len(self.show)):
              self.show[i]+=(m.show[i]-self.show[i])*randint(5000,20000)/10000
                        self.show[i]+=self.change().show[i]*100*randint(5000,20000)/10000
    self.update()
```

In the optimization of smooth functions, the introduction of inertia factors greatly eliminates the dependence of optimization results on the initial group position, achieving good results. But the inertia factor also makes the algorithm more complex, and this problem will be solved in the next generation of rectangular algorithms. The
introduction of inertia factors in the

2.3 third-generation

clustering algorithm may have advantages in certain special situations and can be considered, but in most cases, this algorithm is still too complex and consumes memory. In order to facilitate routine optimization work, it is still necessary to further improve the algorithm.

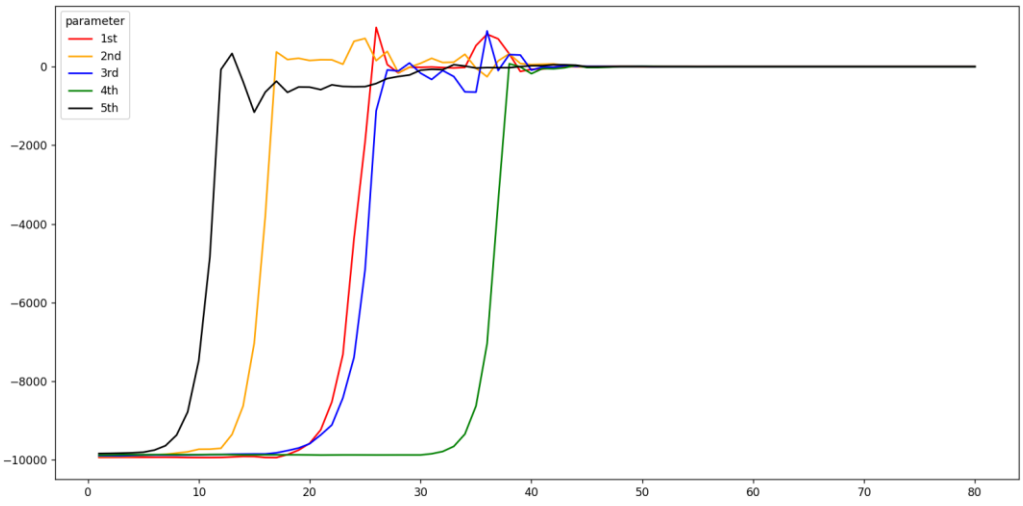
Think about the shortcomings of the first generation clustering algorithm from the root. In the initial group state, when the whole is far away from the optimal solution, the group contraction that occurs synchronously during the movement causes premature convergence of the group, thereby preventing clustering from occurring near the optimal solution.

To solve this problem, the core is to avoid premature convergence of the group and ensure its divergence appropriately. The introduction of inertia factor is one method, while the other can be seen from the statement in the previous section that "from the movement of the group to the optimal individual being less than or equal to before the movement," indicating that the range of random coefficients can be expanded. For example $\in [0.5, 2.5]$, according to geometric probability analysis, it can be concluded that the probability of an individual's distance from the optimal individual increasing $\frac{2.5-2}{2.5-0.5}$is 1/4, ensuring that the group does not converge too early. In practical operation, I do not change the lower limit and set the upper limit to 3. The probability of increasing the distance from an individual to the optimal individual is 2/5, and this algorithm has achieved good results. In fact, when the upper limit is too large, the program may not converge to the optimal solution. Therefore, it is reasonable to take the random factor in the range of 0.5 to 3.

In order to test to what extent the improved clustering algorithm solves the problem of premature convergence, I still used the function introduced in Section 2.1 as the objective function, and set the initialization group at a distance of several hundred units from the optimal solution. The convergence effect was good, and it still had a good effect at a distance of over $^{\text{ten thousand}}$ units from the optimal solution. However, when the distance was raised to the power of 108, the convergence

was not sufficient. So there are two ways to avoid this problem, one is to increase the number of iterations, and the other is hierarchical optimization: consider the optimization parameters $\vec{r}$ as a $\vec{r}$ function of vectors, which can be understood as parameter arrays. The function is a mapping from a vector to a real number, and its meaning is the objective function. So the $f\ (\vec{r})$ search for right optimization can first shift to $f\ (10000\vec{r})$ search for right or $f\ (100\vec{r})$ even waiting optimization. First, determine the approximate range of the optimal solution, and then initialize the list within this large range to find the optimal solution position for fr. $\overrightarrow{r_m}$ When the desired optimal solution is more precise, it can be optimized on the basis of the previous level's optimization results, $f\ (\frac{1}{100}\vec{r})$ even $f\ (\frac{1}{10000}\vec{r})$ equally, in order to make the optimization results more accurate.

The third-generation clustering algorithm has strong applicability. Therefore, I conducted a separate study on the convergence process by plotting it separately. Based on the optimal solution derived from theory, I calculate the distance between each parameter in the convergence process and the theoretical optimal solution, and use this as the vertical axis to plot it with a five color curve. The horizontal axis represents the number of iterations. To test its convergence when the initial group position is far from the optimal solution, I initialized the group at about 10000 units away from the optimal solution. The results are shown below.



From this, it can be seen that for the theoretically feasible optimal solutions (60, 80, 100, 120, 160), the clustering algorithm outputs accurate results and converges quickly. Even if the initial group position is far away from the optimal solution, it can still accurately and quickly converge to the optimal solution, demonstrating good robustness.

Another solution to the problem

of

rapid convergence

based on gradient descent, which leads to the inability to provide the optimal solution, is that after excessive convergence, individuals in the group still have a chance to move.

To this end, I introduced gradient factors and obtained the objective function by solving numerical derivatives. The gradient vectors at each individual are multiplied by the gradient factor and a random number while moving, and then added to the individual. Given that the desired value is a minimum, a negative gradient vector is added.

The gradient calculation method and update method are as follows:

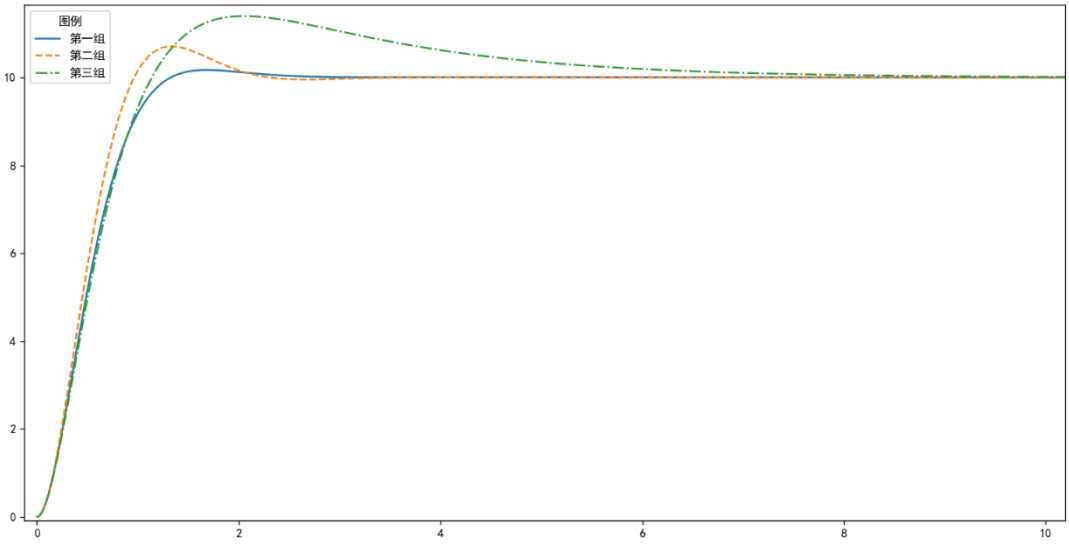
```
def grad(self,f):
    ls=self.show
    grad=[]
    fp=f(ls)
    for k in range(len(ls)):
        ls[k]+=0.001
        grad.append((f(ls)-fp)/0.001)
        ls[k]-=0.001
    return grad
def ran(self,m):
    gr=self.grad(f)
    for i in range(len(self.show)):
        self.show[i]+=(m.show[i]-self.show[i])*randint(5000,30000)/10000+gr[i]/10
```

This improved version can solve the problem to a certain extent, but it has requirements for the objective function, smoothness, etc., and its adaptability is still not as good as the third-generation clustering algorithm. Therefore, it is only used as an improved version, as a possible approach to facing special problems, and is not commonly used by me.

3.1 Using clustering algorithm to optimize PID parameters

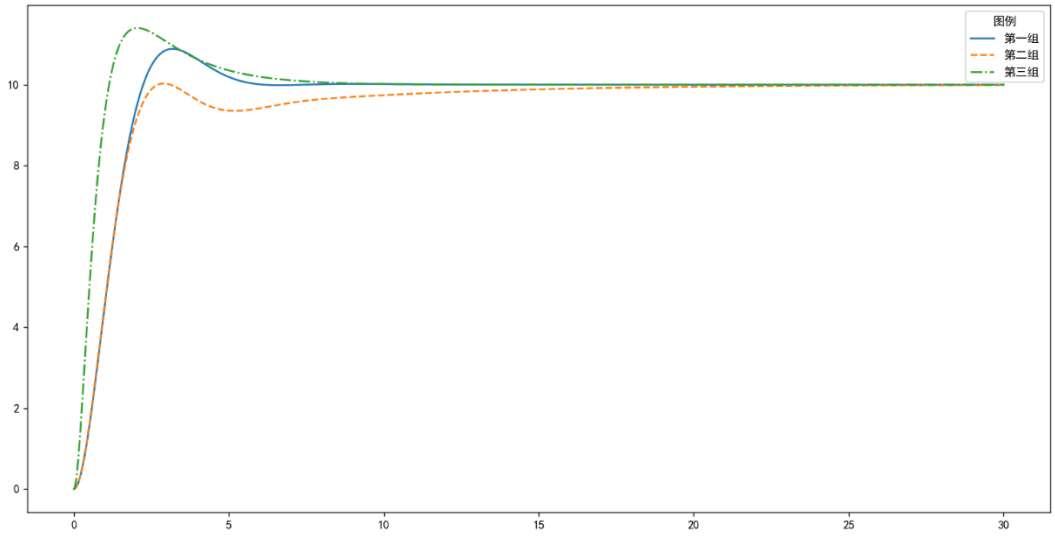
The functions mentioned in the previous text are mathematically smooth. In order to verify the optimization ability of clustering algorithms for more complex functions, we attempted to find some practical optimization problems in engineering for testing.

The application of clustering algorithm in engineering includes the optimization of PID control parameters. The actual situation is to output control force for displacement control under constant resistance disturbance (such as controlling the altitude of a quadcopter aircraft under gravity). Firstly, try to determine the integral gain under the proportional gain (P). Differential gain optimization. The objective function is set to minimize overshoot and achieve the control target around five seconds, with a greater emphasis on the former among the two. Simulate the optimized parameters and obtain the following control image:



The image corresponding to the "first group" is controlled by PID parameters obtained through clustering algorithm optimization. It can be observed that there is almost no overshoot in this control, and the time required to achieve the control target is also very short

In PID, it is uncertain whether a balance is achieved between the square integral of cumulative error and the square integral of cumulative output force and control time. After optimization, the control image is obtained as follows:

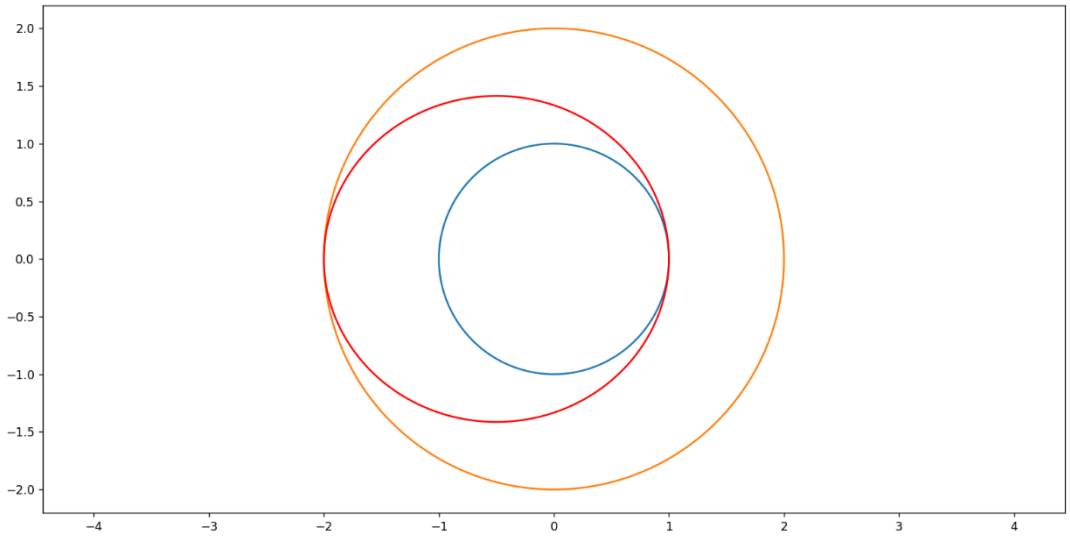


The first and second groups are the outputs of the clustering algorithm. Due to the fact that PID optimal control may not have unique parameter settings, and the objective function design itself may correspond to multiple independent variables with one function value, and the function may not be smooth or continuous, the objective function value of the output result may sometimes be larger and sometimes smaller. Therefore, I chose two sets of solutions with smaller objective function values, and their corresponding control images are shown in the above figure. The effect is still relatively ideal.

3.3 Searching for dual impulse transfer in coplanar circular orbits

Optimization based on clustering algorithm can also be used to design optimal transfer trajectories for dual impulse transfer of coplanar orbits. According to Hohmann's theory, the optimal transfer orbit should be tangent to the initial orbit and the target orbit at the periapsis and apogee points, respectively. The clustering algorithm can be used to verify.

Taking a circular orbit with an initial radius of 1 unit and a target of 2 units as an example for optimization. After multiple optimizations, the optimal result is as follows:



This is undoubtedly consistent with Hohmann's theory.

3.3 Equation solving

clustering algorithm can be used for equation solving, such as for F (x)=0, the objective function fx can be designed to be the absolute value of F (x), or fx= ^{F2}（x）， Solving the original equation is equivalent to finding the minimum value of the objective function.

For example

, the real root objective function for

solving ^{x3}-^{2x2}-^{6} x-80=0 is designed as | ^{x3}-^{2x2}-6 x-80 |. The output result is approximately 4.13358, with an objective function value on the order of $^{10-16}$ to $^{10-13}$, which is very close to the true root. This is the case where the equation has a unique real root. For equations with multiple real roots, changing the initial group state can lead to convergence to different real roots.

The clustering algorithm can also be used to solve the intersection points of curved surfaces, that is, to find the real solutions of the system of equations, such as finding C: ^{x2}+^{y2}=^{4} and l:

The √2intersection point of x+y-2=0 can be designed as the objective function f (x, y)=^{x2}+^{y2}=^{4}. The result is approximately (1.414213, 1.414213), which is very close to (√2， √2). Returning to the original equation, it can be proved that this is the actual intersection point of the two lines.

Generally, for $_{Fj}$ ($_{x1}$, $_{x2}$,... $_{xn}$) (j=1,2,..., k), if it is necessary to find their intersection points, the objective function can be designed as follows:

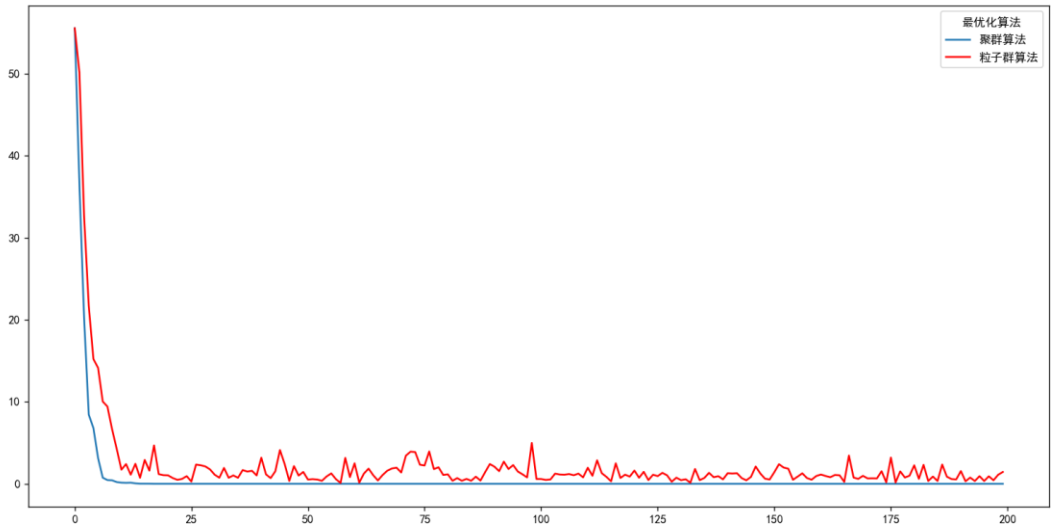
$$f(x1,\dots,xn)=\sum_{j=1}^{k}\left|F_j(x1,\dots,xn)\right|$$

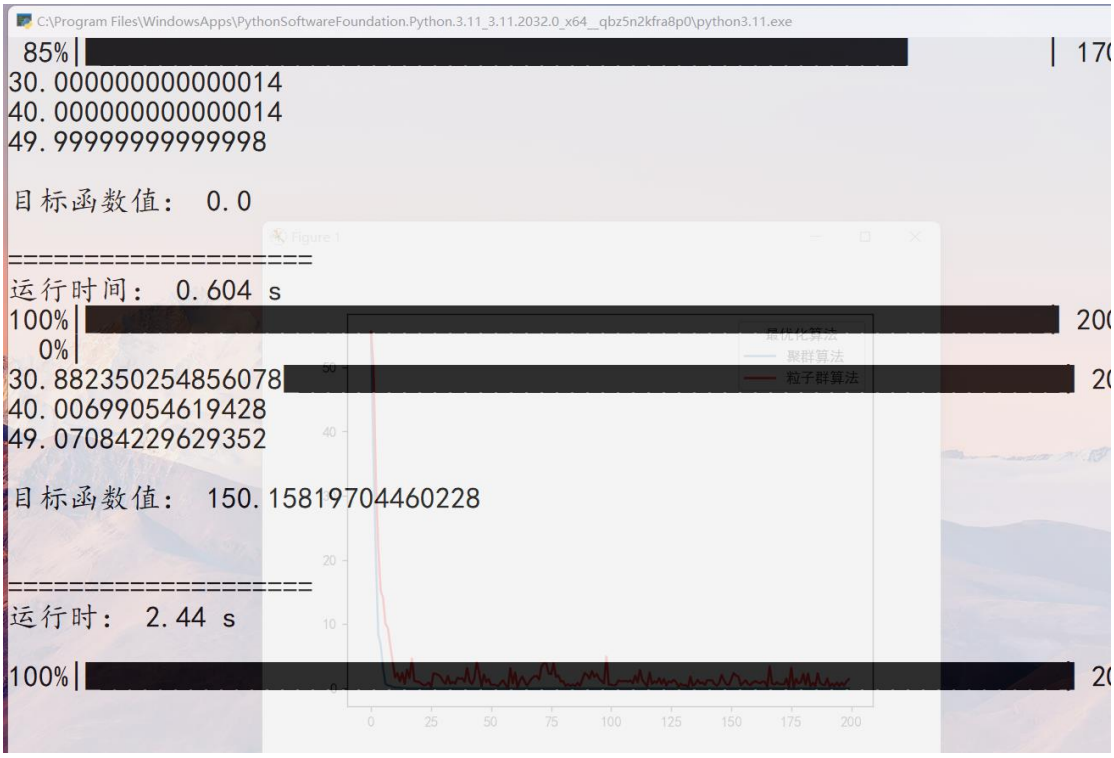
perhaps

$$f(x1,\dots,xn)=\sum_{j=1}^{k}F_j^2(x1,\dots,xn)$$

So finding intersection points is reduced to optimizing the objective function.

In the end, we use an example to compare the clustering algorithm with the particle swarm algorithm, considering a scenario in navigation positioning. Assuming that the distances from a point to four positioning points: (0,10,0), (0,-10,0), (10,0,0), and (0,0,10) are approximately 65.5744, 76.8115, 67.0821, and 64.0312, respectively, we can determine the intersection points of four spherical surfaces with four centers and corresponding distances as radii. In fact, this point is designed as (30, 40, 50). Due to the fact that the centers of the four balls are not coplanar, the intersection point is unique. We use this to test the performance of clustering algorithm and particle matrix algorithm. The method described in section 2.3 yields the following results, where the horizontal axis represents the number of iterations and the vertical axis represents the Euclidean distance from the output result to the pre designed point:





In the screenshot of the console, the upper part is the output result of the clustering algorithm, and the lower part is the output result of the particle swarm algorithm.

From the results, the clustering algorithm has higher accuracy and converges faster and more stably during the process.