**по лабораторной работе № 4**

**Дисциплина: Оптимизация вычислительных процессов**

# Тема: Алгоритм метода рой частиц

Выполнил:

Студент 46/2 группы

Ролдугин А.С.

Преподаватель:

Климец А.А.

# Цель работы:

Реализовать на практике Метод поиска с использованием производных – Градиентный спуск.

**1. Алгоритм метода роя частиц**

Алгоритм роя частиц относится к стохастическим алгоритмам с **одновременным поиском решения сразу по всей области поиска.**

Идея алгоритма была частично заимствована из исследований поведения скоплений животных (косяков рыб, стай птиц и т. п.).

Алгоритм роя частиц - представим n-мерное пространство (область поиска), в котором рыщут частицы (агенты алгоритма). В начале частицы разбросаны случайным образом по всей области поиска, и каждая частица имеет случайный вектор скорости. В каждой точке, где побывала частица, рассчитывается значение целевой функции. При этом каждая частица запоминает лучшее значение целевой функции она лично нашла, а также каждая частица знает где расположена точка, являющаяся лучшей среди всех точек, которые разведали частицы. На каждой итерации частицы корректируют свою скорость (модуль и направление), чтобы, с одной стороны, быть поближе к лучшей точке, которую частица нашла сама "ностальгия", и, в то же время, приблизиться к точке, которая в данный момент является глобально лучшей. Через некоторое количество итераций частицы должны собраться вблизи наиболее хорошей точки, вблизи глобального экстремума. Самое интересное в алгоритме — это коррекция скорости, именно от этого шага зависит сходимость алгоритма.

**2. Текст программы «Алгоритм метода рой частиц»**

Подключим функцию оптимизатора pso из библиотеки pyswarm :

**from**  **pyswarm**  **import**  pso

Определим минимизируемую целевую функцию myfunction (x) .

Функция будет возвращать единственное минимизированное скалярное значение.

 Исследуем работу алгоритма роя частиц на примере оптимизации banana-функции:

**def**  banana ( x ):

x1 = x [ 0 ]

x2 = x [ 1 ]

**return**  x1 \*\* 4 - 4 \* x2 \* x1 \*\* 2 + x2 \*\* 2 + x1 \*\* 2 - 4 \* x1 + 10

Определим пределы входных переменных, в которых оптимизатору разрешен поиск в lb и ub, которые обозначают нижнюю и верхнюю границы соответственно:

lb = [ - 3 , - 1 ]

ub = [ 2 , 6 ]

Теперь все определено для запуска pso, поэтому вызываем оптимизатор:

xopt, fopt = pso(banana, lb, ub, f\_ieqcons=con)

f\_ieqcons - сообщает подпрограмме, что существует единственная функция ограничения.

После завершения PSO возвращает два объекта:

1) оптимальные входные значения

2) оптимальное целевое значение.

Синтаксис вызова для PSO  настраивается и определяется следующим образом:

pso ( func , lb , ub , ieqcons = [], f\_ieqcons = None , swarmsize = 100 , omega = 0,5 , phip = 0,5 , maxiter = 100 , minstep = 1e -8 , minfunc = 1e-8 )

где минимальные необходимые входные аргументы:

func : function - минимизируемая функция;

lb : array - нижние границы проектной переменной;

ub : array - верхние границы проектной переменной;

f\_ieqcons  - функция возвращает одномерный массив, в котором каждый элемент должен быть больше или равен 0,0 в успешно оптимизированной задаче;

swarmsize : int - количество частиц в рое (по умолчанию: 100);

omega : scalar - коэффициент масштабирования скорости частиц (по умолч: 0,5);

phip : scalar - коэффициент масштабирования для поиска вдали от наиболее известного положения частицы (по умолчанию: 0,5);

Параметры omega и phip позволяют контролировать, насколько близко частицы удаляются от своего самого известного положения. Они могут принимать любое скалярное значение, но лучше всего подходят значения от 0 до 1.

maxiter : int - максимальное количество итераций для поиска роя (по умолч: 100);

minstep : scalar - минимальный размер шага лучшей позиции роя до прекращения поиска (по умолчанию: 1e-8);

minfunc : scalar - минимальное изменение наилучшего целевого значения роя до завершения поиска (по умолчанию: 1e-8);

Параметры maxiter , minstep и minfunc используются, чтобы сообщить рою, когда следует прекратить поиск.

**from** functools **import** partial

**import** numpy **as** np

**def** \_obj\_wrapper**(**func**,** x**):**

**return** func**(**x**)**

**def** \_is\_feasible\_wrapper**(**func**,** x**):**

**return** np**.all(**func**(**x**)>=**0**)**

**def** \_cons\_none\_wrapper**(**x**):**

**return** np**.**array**([**0**])**

**def** \_cons\_ieqcons\_wrapper**(**ieqcons**,** x**):**

**return** np**.**array**([**y**(**x**)** **for** y **in** ieqcons**])**

**def** \_cons\_f\_ieqcons\_wrapper**(**f\_ieqcons**,** args**,** kwargs**,** x**):**

**return** np**.**array**(**f\_ieqcons**(**x**))**

**def** pso**(**func**,** lb**,** ub**,** ieqcons**=[],** f\_ieqcons**=None,**

swarmsize**=**100**,** omega**=**0.5**,** phip**=**0.5**,** phig**=**0.5**,** maxiter**=**100**,**

minstep**=**1e-8**,** minfunc**=**1e-8**,** processes**=**1**,**

particle\_output**=True):**

**assert** **len(**lb**)==len(**ub**),** 'Нижняя и верхняя границы '

**assert** **hasattr(**func**,** '\_\_call\_\_'**),** 'Недействительный дескриптор функции'

lb **=** np**.**array**(**lb**)**

ub **=** np**.**array**(**ub**)**

**assert** np**.all(**ub**>**lb**),** 'Все значения верхней границы должны быть больше значений нижней границы'

vhigh **=** np**.abs(**ub **-** lb**)**

vlow **=** **-**vhigh

# Инициализируем целевую функцию

obj **=** partial**(**\_obj\_wrapper**,** func**)**

# BИнициализируем алгоритм рой частиц

S **=** swarmsize

D **=** **len(**lb**)** # количество измерений каждой частицы

x **=** np**.**random**.**rand**(**S**,** D**)** # положения частиц

v **=** np**.**zeros\_like**(**x**)** # скорости частиц

p **=** np**.**zeros\_like**(**x**)** # лучшие позиции частиц

fx **=** np**.**zeros**(**S**)** # текущие значения функции частиц

fs **=** np**.**zeros**(**S**,** dtype**=bool)** # выполнимость каждой частицы

fp **=** np**.**ones**(**S**)\***np**.**inf # лучшие значения функции частиц

g **=** **[]** # лучшая позиция роя

fg **=** np**.**inf # начальное значение наилучшей позиции роя

# Инициализируем положение частицы

x **=** lb **+** x**\*(**ub **-** lb**)**

# Вычислим цель и ограничения для каждой частицы

**if** processes **>** 1**:**

fx **=** np**.**array**(**mp\_pool**.map(**obj**,** x**))**

fs **=** np**.**array**(**mp\_pool**.map(**is\_feasible**,** x**))**

**else:**

**for** i **in** **range(**S**):**

fx**[**i**]** **=** obj**(**x**[**i**,** **:])**

fs**[**i**]** **=** is\_feasible**(**x**[**i**,** **:])**

# Сохранить лучшую позицию частицы

i\_update **=** np**.**logical\_and**((**fx **<** fp**),** fs**)**

p**[**i\_update**,** **:]** **=** x**[**i\_update**,** **:].**copy**()**

fp**[**i\_update**]** **=** fx**[**i\_update**]**

# Обновить лучшую позицию роя

i\_min **=** np**.**argmin**(**fp**)**

**if** fp**[**i\_min**]** **<** fg**:**

fg **=** fp**[**i\_min**]**

g **=** p**[**i\_min**,** **:].**copy**()**

**else:**

# Задаем Отправную начальная точка роя частиц

g **=** x**[**0**,** **:].**copy**()**

# Инициализируем скорость частицы

v **=** vlow **+** np**.**random**.**rand**(**S**,** D**)\*(**vhigh **-** vlow**)**

# Итерируем до тех пор, пока не будет выполнен критерий завершения

it **=** 1

**while** it **<=** maxiter**:**

rp **=** np**.**random**.**uniform**(**size**=(**S**,** D**))**

rg **=** np**.**random**.**uniform**(**size**=(**S**,** D**))**

# Обновление скорости частиц

v **=** omega**\***v **+** phip**\***rp**\*(**p **-** x**)** **+** phig**\***rg**\*(**g **-** x**)**

# Обновление положения частиц

x **=** x **+** v

# Исправление для связанных нарушений гарниц

maskl **=** x **<** lb

masku **=** x **>** ub

x **=** x**\*(~**np**.**logical\_or**(**maskl**,** masku**))** **+** lb**\***maskl **+** ub**\***masku

# Обновление цели и ограничения

**if** processes **>** 1**:**

fx **=** np**.**array**(**mp\_pool**.map(**obj**,** x**))**

fs **=** np**.**array**(**mp\_pool**.map(**is\_feasible**,** x**))**

**else:**

**for** i **in** **range(**S**):**

fx**[**i**]** **=** obj**(**x**[**i**,** **:])**

fs**[**i**]** **=** is\_feasible**(**x**[**i**,** **:])**

# Сохраняем лучшую позицию частицы

i\_update **=** np**.**logical\_and**((**fx **<** fp**),** fs**)**

p**[**i\_update**,** **:]** **=** x**[**i\_update**,** **:].**copy**()**

fp**[**i\_update**]** **=** fx**[**i\_update**]**

# Сравните лучшую позицию роя с текущей позицией

i\_min **=** np**.**argmin**(**fp**)**

**if** fp**[**i\_min**]** **<** fg**:**

**if** debug**:**

**print(**' Новое лучшее для роя на итерациях {:}: {:} {:}'\

**.format(**it**,** p**[**i\_min**,** **:],** fp**[**i\_min**]))**

p\_min **=** p**[**i\_min**,** **:].**copy**()**

stepsize **=** np**.**sqrt**(**np**.sum((**g **-** p\_min**)\*\***2**))**

**if** np**.abs(**fg **-** fp**[**i\_min**])** **<=** minfunc**:**

**print(**'Остановка поиска: лучшая цель роя изменилась менее чем {:}'\

**.format(**minfunc**))**

**if** particle\_output**:**

**return** p\_min**,** fp**[**i\_min**],** p**,** fp

**else:**

**return** p\_min**,** fp**[**i\_min**]**

**elif** stepsize **<=** minstep**:**

**print(**'Остановка поиска: изменение лучшей позиции роя меньше чем {:}'\

**.format(**minstep**))**

**if** particle\_output**:**

**return** p\_min**,** fp**[**i\_min**],** p**,** fp

**else:**

**return** p\_min**,** fp**[**i\_min**]**

**else:**

g **=** p\_min**.**copy**()**

fg **=** fp**[**i\_min**]**

**if** debug**:**

**print(**' Лучше всего после итерации {:}: {:} {:}'**.format(**it**,** g**,** fg**))**

it **+=** 1

**print(**'Остановка поиска: достигнуто максимальное количество итераций --> {:}'**.format(**maxiter**))**

**if** **not** is\_feasible**(**g**):**

**print(**"Оптимизация НЕ ДОСТИГНУТА !!!!!!!"**)**

**if** particle\_output**:**

**return** g**,** fg**,** p**,** fp

**else:**

**return** g**,** fg

Рисунок 1 – Текст программы «Алгоритм метода роя частиц»

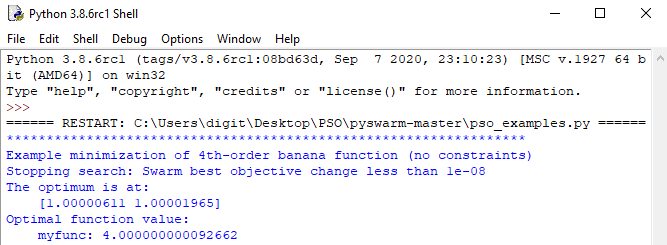


Рисунок 2 – Результат работы прогарммы

**Вывод:**

В ходе выполненной работы средствами ЯП Python исследован на практике Алгоритм метода рой частиц и определены условия оптимальности полученного решения.