Energy valley optimization

Васильченко О.В.

гр. 9301

1. Введение

Данная работа посвящена исследованию и программной реализации метаэвристрического алгоритма оптимизации "Energy valley optimizer", представленного Mahdi Azizi в соответствующей статье для журнала Nature.

Текущая секция работы является введением и содержит краткую информацию о содержании и стуктуре проведенной работы.

Вторая секция работы посвящена описанию исследуемого алгоритма и тестовым функциям, на которых исследуемый алгоритм будет испытан для оценки точности и корректности его работы. Для исследуемых функций приведены формулы, информация о минимумах и их координатах соответственно.Для исследуемого алгоритма приведен и описан псевдокод его работы.

Третья секция работы "Результаты" содержит построенные графики поиска исследуемым алгоритмом на пространстве выбранных функций. Код программы для Python приведен в приложении к данной работе.

Четвертая секция "Выводы" включает таблицу с информацией о работе алгоритма для каждой из рассмотренных функций. Полученные результаты описаны и разобраны.

2. Алгоритм и тестовые функции

2.1. Описание алгоритма

В первую очередь во время работы алгоритма оптимизации EVO выполняется процесс инициализации. В рамках этого процесса кандидатами решений (x_i) считаются частицы с различным уровнем устойчивости внутри пространства поиска, которое считается некоторой частью большей области. n - количество таких частиц, а d - размерность рассматриваемой функции.

На следующем этапе алгоритма определяется граница обогащения $(Enrichment\ boundary,\ EB)$ для частиц, которая используется для рассмотрения различий между нейтронно-богатыми и нейтронно-бедными частицами. С этой целью оценка целевой функции для каждой из частиц определяется как уровень нейтронного обогащения $(Neutron\ Enrichment\ Level,\ NEL)$ частиц.

$$EB = \frac{\sum_{i=1}^{1} \mathrm{NEL}_i}{n}, \quad i = 1, 2, \ldots n.$$

Далее определяются уровни стабильности частиц на основании целевой функции.

$$\mathrm{SL}_i = \frac{\mathrm{NEL}_i - \mathrm{BS}}{\mathrm{WS} - \mathrm{BS}}, \quad i = 1, 2, \ldots n.$$

В основном цикле поиска EVO, если уровень нейтронного обогащения частицы выше, чем обогащение ($NEL_i > EB$), предполагается, что частица имеет большее соотношение N/Z, поэтому выполняется процесс распада с использованием альфа, бета или гамма-схемы. В связи с этим генерируется случайное число в диапазоне $[0,\ 1]$, которое имитирует границу стабильности (SB). Если уровень устойчивости частицы выше границы устойчивости ($SLi > SB$), альфа- и гамма-распады считаются имеющими место, поскольку эти два распада вероятны для более тяжелых частицы с более высоким уровнем стабильности.

Испускаемые лучи являются переменными решения в кандидате решения, которые удаляются и заменяются лучами в частице или кандидате с наилучшим уровнем устойчивости (X_{BS}). Математически эти аспекты формулируются следующим образом:

$$X_i^{\mathrm{New1}} = X_i \left(X_{\mathrm{BS}} \left(x_i^j \right) \right), \quad \begin{cases} i = 1, 2, \ldots, n. \\ j = \text{AlphaIndex II.} \end{cases},$$

где X_i^{New1} новая сгенерированная частица в пространстве, X_i - текущая позиция вектора i-ой частицы (кандидата решения) в пространстве (области решения),X_{BS} - вектор положения частицы с наилучшим уровнем стабильности, x_i^j - j-ая переменная решения (или излучаемый луч).

Фотоны в частицах являются переменными решения (кандидатами на решение), которые удаляются и заменяются соседней частицей или кандидатом (X_{Ng}), который имитирует взаимодействие возбужденных частиц с другими частицами или даже магнитными полями.

$$D_i^k = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}, \quad \begin{cases} i = 1, 2, \ldots, n. \\ k = 1, 2, \ldots, n-1. \end{cases}$$

Где D_i^k - общее расстояние между i-й частицей и k-й соседней частицей, а выражения в скобках под корнем обозначают координаты частиц в пространстве поиска. После этого, процесс обновления позиции для генерации второго кандидата решения на текущем этапе осуществляется следующим образом:

$$X_i^{\text{New2}} = X_i \left(X_{\text{Ng}} \left(x_i^j \right) \right), \quad \begin{cases} i = 1, 2, \ldots, n. \\ j = \text{GammaIndex II}. \end{cases}$$

Где X_i^{New2} новая сгенерированная частица в пространстве, X_i - текущая позиция вектора i-ой частицы (кандидата решения) в пространстве (области решения), X_{Ng} позиция вектора соседней цастицы вокруг i-ой частицы, x_i^j - j-я переменная решения или испущенный фотон.

Если уровень стабильности частицы ниже границы стабильности ($SL_i \leq SB$), считается, что произошел бета-распад. В связи с этим для частиц проводится процесс обновления позиции, в ходе которого осуществляется контролируемое движение к частице или кандидату с наилучшим уровнем стабильности (X_{BS}) и центру частиц (X_{CP}). Эти аспекты алгоритма имитируют тенденцию частиц к достижению полосы стабильности, в которой большинство известных частиц располагаются вблизи этой полосы, и большинство из них имеют более высокие уровни стабильности и вычисляются следующим образом:

$$X_{cp} = \frac{\sum_{i=1}^{1} \text{X}_i}{n}, \quad i = 1, 2, \ldots n.$$

$$X_i^{\text{New1}} = X_i + \frac{(r_1 \times X_{\text{BS}} - r_2 \times X_{\text{CP}})}{\text{SL}_i}, \quad i = 1, 2, \ldots n.$$

Где X_i^{New1} и X_i это векторы предстоящего и текущего положения i-й частицы (кандидата на решение) в пространстве поиска, X_{BS} - вектор положения частицы с наилучшей стабильностью, X_{CP} - вектор расположения центра частиц, SL_i - уровень стабильность i-ой частицы, $r1$ и $r2$ два случайных числа в промежутке $[0,\ 1]$ определяющие движение частицы.

Для того чтобы улучшить качество исследования алгоритма, для частиц, использующих бета-распад, проводится другой процесс обновления позиции, в котором контролируемое движение к частице или кандидату с наилучшим уровнем стабильности (X_{BS}) и соседней частице или кандидату (X_{Ng}) выполняется, при этом уровень стабильности частицы не влияет на процесс движения. Математически этот процесс формулируются следующим образом:

$$X_i^{\text{New2}} = X_i + (r_3 \times X_{\text{BS}} - r_4 \times X_{\text{Ng}}), \quad i = 1, 2, \ldots n.$$

Если уровень нейтронного обогащения частицы ниже границы обогащения ($NEL_i \leq EB$), предполагается, что частица имеет меньшее отношение N/Z, поэтому частица стремится к захвату электронов или испусканию позитронов, чтобы переместиться в полосу стабильности. В связи с этим случайное движение в пространстве поиска определяется для учета этих видов движений следующим образом:

$$X_i^{\text{New1}} = X_i + r_1, \quad i = 1, 2, \ldots n.$$

В конце основного цикла EVO для каждой из частиц генерируется только два новых вектора положения X_i^{New1} и X_i^{New2}, если уровень обогащения частицы выше границы обогащения, а для частицы с более низким уровнем обогащения в качестве нового вектора положения генерируется только X_i^{New}. В каждом состоянии вновь сгенерированные векторы объединяются с текущей популяцией, и лучшие частицы участвуют в следующем цикле поиска алгоритма. Для переменных решения, выходящих за пределы заданных верхних и нижних границ, определяется маркер нарушение границы, а в качестве критерия завершения может использоваться максимальное количество оценок целевой функции или максимальное количество итераций.

Algorithm 1: Energy Valley Optimizer (EVopt)

Входные данные:
f ; /* Целевая функция */
d ; /* Размерность проблемы */
n ; /* Число частиц */
T ; /* Максимальная итерация */
$[l, u]$; /* Границы начального распределения частиц */
Результат:
x_{min} ; /* Точка минимума */
f_{min} ; /* Значение функции в $x_{\min}$ */
neval ; /* Число вычислений целевой функции */
Initialization:
VarMin $\leftarrow l \cdot \mathbf{1}_d$, VarMax $\leftarrow u \cdot \mathbf{1}_d$;
Particles $\leftarrow$ rand$(n, d) \cdot$ (VarMax $-$ VarMin) $+$ VarMin;
NELs $\leftarrow \{f(\text{Particles}_i) \mid i = 1, \ldots, n\}$;
neval $\leftarrow n$;
bestNEL, bestIdx $\leftarrow$ min(NELs);
worstNEL $\leftarrow$ max(NELs);
$x_{\min} \leftarrow$ Particles[bestIdx];
Main Loop:
for *Iter* $\leftarrow 1$ **to** T **do**
  Xnew $\leftarrow$ [], Xnew1 $\leftarrow$ [], Xnew2 $\leftarrow$ [];
  for $i \leftarrow 1$ **to** n **do**
    enrichmentBound $\leftarrow$ mean(NELs);
    $\text{NEL}_i \leftarrow$ NELs[i];
    if $NEL_i >$ *enrichmentBound* **then**
      stabilityBound $\leftarrow$ rand();
      stabilityLevel $\leftarrow$ (NEL_i $-$ bestNEL)/(worstNEL $-$ bestNEL);
      if *stabilityLevel* $>$ *stabilityBound* **then**
        alphaIndex1 $\leftarrow$ randint(1, d);
        alphaIndex2 $\leftarrow$ randint(1, alphaIndex1);
        Xnew1.$append$(Particles[i]);
        Xnew1[end, alphaIndex2] $\leftarrow x_{\min}$[alphaIndex2];
        gammaIndex1 $\leftarrow$ randint(1, d);
        gammaIndex2 $\leftarrow$ randint(1, gammaIndex1);
        Dist $\leftarrow$ distance(Particles[i], Particles);
        nearestIdx $\leftarrow$ argmin(Dist[Dist > 0]);
        Xng $\leftarrow$ Particles[nearestIdx];
        if *Xng* $\neq \emptyset$ **then**
          Xnew2.$append$(Particles[i]);
          Xnew2[end, gammaIndex2] $\leftarrow$ Xng[gammaIndex2];
        end
      end
      else
        Xcp $\leftarrow$ mean(Particles);
        Xnew1.$append$(Particles[i] $+$ rand $\cdot$ $x_{\min}$ $-$ rand $\cdot$ Xcp);
        Dist $\leftarrow$ distance(Particles[i], Particles);
        nearestIdx $\leftarrow$ argmin(Dist[Dist > 0]);
        Xng $\leftarrow$ Particles[nearestIdx];
        Xnew2.$append$(Particles[i] $+$ rand $\cdot$ $x_{\min}$ $-$ rand $\cdot$ Xng);
      end
      Xnew1[end] $\leftarrow$ clip(Xnew1[end], VarMin, VarMax);
      Xnew2[end] $\leftarrow$ clip(Xnew2[end], VarMin, VarMax);
    end
    else
      Xnew.$append$(Particles[i] $+$ rand);
      Xnew[end] $\leftarrow$ clip(Xnew[end], VarMin, VarMax);
    end
  end
  Particles $\leftarrow$ [Particles; Xnew; Xnew1; Xnew2];
  NELs $\leftarrow [f(\text{Particles}_i) \mid i = 1, \ldots, \text{size(Particles, 1)}]$;
  order $\leftarrow$ argsort(NELs);
  NELs $\leftarrow$ NELs[order[: n]];
  Particles $\leftarrow$ Particles[order[: n], :];
  bestNEL $\leftarrow$ min(NELs), worstNEL $\leftarrow$ max(NELs);
  $x_{\min} \leftarrow$ Particles[argmin(NELs)];
  $f_{\min} \leftarrow$ bestNEL;
end
return x_{min}, f_{min}, *neval*

2.2. Тестовые проблемы

2.2.1. Проблеммы с множественным локальным минимумом

Функции этой категории характеризуются наличием множества локальных минимумов, что делает оптимизацию сложной для алгоритмов, основанных на градиентных методах, которые могут легко застрять в локальных минимумах. Эвристические и метаэвристические подходы, такие как генетические алгоритмы, оптимизация роя частиц и моделируемый отжиг, обычно лучше подходят для таких функций, поскольку они исследуют пространство решений более глобально.

Ackley Function Функция `Ackley` широко используется для тестирования алгоритмов оптимизации благодаря множеству локальных минимумов и единственному глобальному минимуму. Высокая

частота колебаний мешает градиентным методам, в то время как метаэвристические алгоритмы превосходят их благодаря способности исследовать различные области пространства решений.

$$f(x) = -20\exp\left(-0.2\sqrt{\frac{1}{d}\sum_{i=1}^{d}x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(2\pi x_i)\right) + 20 + e, \tag{1}$$

$$-5 \le x_i \le 5, \quad x_{\text{min}} = (0, \ldots, 0), \quad f_{\text{min}} = 0.$$

Eggholder Function Функция `Eggholder` известна своим сложным ландшафтом с многочисленными крутыми долинами и хребтами. Резкие переходы градиента делают ее сложной для квазиградиентных методов. Такие алгоритмы, как дифференциальная эволюция и роевые методы, часто оказываются более эффективными.

$$f(x_1, x_2) = -(x_2 + 47)\sin\left(\sqrt{|x_2 + x_1/2 + 47|}\right) - x_1\sin\left(\sqrt{|x_1 - (x_2 + 47)|}\right), \tag{2}$$

$$-512 \le x_1, x_2 \le 512, \quad x_{\text{min}} \approx (512, 404.2319), \quad f_{\text{min}} \approx -959.6407.$$

2.2.2. Проблемы "чаши"

Эти функции гладкие и унимодальные, с одним глобальным минимумом, который лежит в параболической области. Они хорошо подходят для градиентных и квазиньютоновских методов благодаря своей выпуклой природе.

Sphere Function `Sphere` это простой квадратичный тест, используемый для оценки алгоритмов оптимизации. Его гладкая поверхность делает его идеальным для алгоритмов, использующих информацию о градиенте.

$$f(x) = \sum_{i=1}^{d} x_i^2, \tag{3}$$

$$-5.12 \le x_i \le 5.12, \quad x_{\text{min}} = (0, \ldots, 0), \quad f_{\text{min}} = 0.$$

Trid Function Функция `Trid` создана для того, чтобы бросить вызов алгоритмам оптимизации с неквадратичной формой чаши. Хотя она остается унимодальной, нелинейность требует квазиньютоновских методов или продвинутых градиентных оптимизаторов для эффективного сближения.

$$f(x) = \sum_{i=1}^{d}(x_i - 1)^2 - \sum_{i=2}^{d} x_i x_{i-1}, \tag{4}$$

$$-d^2 \le x_i \le d^2, \quad x_{\text{min}_i} = (i(d + 1 - i)) \text{ for all } i = 1, 2 \ldots d, \quad f_{\text{min}} = -d(d + 4)(d - 1)/6.$$

2.2.3. Функции "плато"

Эти функции имеют плоские области или плато, что может замедлить работу алгоритмов оптимизации. Линейные методы и простой градиентный спуск часто оказываются здесь неэффективными, в то время как эвристические подходы, использующие случайную выборку или имитацию отжига, могут быть более эффективными.

McCormick Function Функция `McCormick` - это двумерная функция с относительно простой поверхностью, но невыпуклыми свойствами. Плато может ввести в заблуждение градиентные алгоритмы, но эвристические методы справляются неплохо.

$$f(x_1, x_2) = \sin(x_1 + x_2) + (x_1 - x_2)^2 - 1.5x_1 + 2.5x_2 + 1, \tag{5}$$

$$-1.5 \le x_1 \le 4, -3 \le x_2 \le 4, \quad x_{\text{min}} \approx (-0.54719, -1.54719), \quad f_{\text{min}} \approx -1.9133.$$

Booth Function Функция `Booth` широко используется в оптимизации для тестирования алгоритмов с двумя переменными. Хотя функция имеет простую структуру, области плато могут замедлить сходимость для линейных или простых градиентных методо

$$f(x_1, x_2) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2, \tag{6}$$

$$-10 \leq x_1, x_2 \leq 10, \quad x_{\text{min}} = (1, 3), \quad f_{\text{min}} = 0.$$

2.2.4. Проблемы "долины"

Эти функции имеют узкие долины, в которых трудно найти глобальный минимум. Градиентные алгоритмы могут колебаться в пределах долины, прежде чем сходятся. Алгоритмы с адаптивным размером шага или гибридные метаэвристические методы могут работать лучше.

Rosenbrock Function The `Rosenbrock`, также известная как функция Банана, является классическим эталоном с изогнутой долиной, ведущей к глобальному минимуму. Градиентные методы часто оказываются неэффективными из-за плоскости вдоль оси долины, что требует применения продвинутых импульсных методов.

$$f(x) = \sum_{i=1}^{d-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right], \tag{7}$$

$$-5 \leq x_i \leq 10, \quad x_{\text{min}} = (1, \ldots, 1), \quad f_{\text{min}} = 0.$$

Six-Hump Camel Function Функция `Six-Hump Camel` представляет собой сложный двумерный тестовый пример с несколькими локальными минимумами. Для успешного решения проблемы на такой функции, вероятно, потребуются алгоритмы с возможностями глобального поиска, такие как имитация отжига или генетические алгоритмы, чтобы не застревать в локальных минимумах.

$$f(x_1, x_2) = 4x_1^2 - 2.1x_1^4 + \frac{x_1^6}{3} + x_1x_2 - 4x_2^2 + 4x_2^4, \tag{8}$$

$$-3 \leq x_1 \leq 3, -2 \leq x_2 \leq 2, \quad x_{\text{min}} \approx (\pm 0.0898, \mp 0.7126), \quad f_{\text{min}} \approx -1.0316.$$

2.2.5. Проблемы "хребтов и падений"

Функции этой категории характеризуются резкими изменениями градиента, что делает их труднопроходимыми для алгоритмов оптимизации. Метаэвристические методы, выполняющие широкую выборку, подходят лучше, чем традиционные градиентные подходы.

Easom Function Функция `Easom` имеет один резкий глобальный минимум, окруженный крутыми гребнями. Эта функция бросает вызов как градиентным, так и квазиградиентным алгоритмам, требуя стратегий глобального поиска для эффективного нахождения минимума.

$$f(x_1, x_2) = -\cos(x_1)\cos(x_2)\exp\left(-(x_1 - \pi)^2 - (x_2 - \pi)^2\right), \tag{9}$$

$$-100 \leq x_1, x_2 \leq 100, \quad x_{\text{min}} = (\pi, \pi), \quad f_{\text{min}} = -1.$$

Michalewicz Function Функция `Michalewicz` - это мультимодальная проблема, зависящая от параметров d (размерность) и m (резкость минимумов). Крутые спады и узкие минимумы затрудняют работу большинства детерминированных методов, делая метаэвристические алгоритмы, такие как оптимизация муравьиной колонии или моделированный отжиг, более эффективными.ментальные методы.

$$f(x) = -\sum_{i=1}^{d} \sin(x_i)\left[\sin\left(\frac{ix_i^2}{\pi}\right)\right]^{2m}, \tag{10}$$

$$0 \leq x_i \leq \pi, \quad x_{\text{min}} \text{ varies with } d, m.$$

3. Результаты

Алгоритм показал себя достаточно хорошо при решении различных задач оптимизации, погрешности в большинстве случаев минималь, однако количество обращений к целевой функции значительное (по сравнение с прочими, более простыми методами оптимизации). Наименее эффективно алгоритм показал себя при решении задач на функциях с множественными локальными минимумами. Так, наибольшую ошибку алгоритм продемонстрировал при решении задачи оптимизации для функции Eggholder.

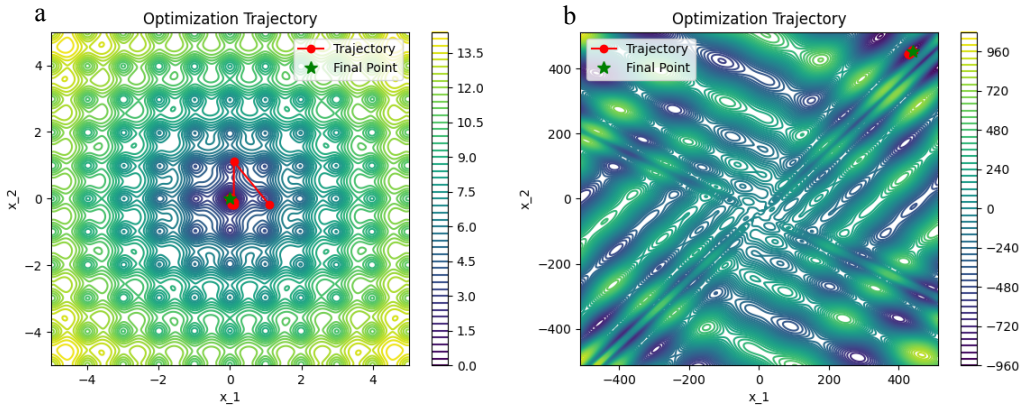

Рис. 1: Траектория поиска для проблем с множественным локальным минимумом. a - `Ackley`, b - `Eggholder`

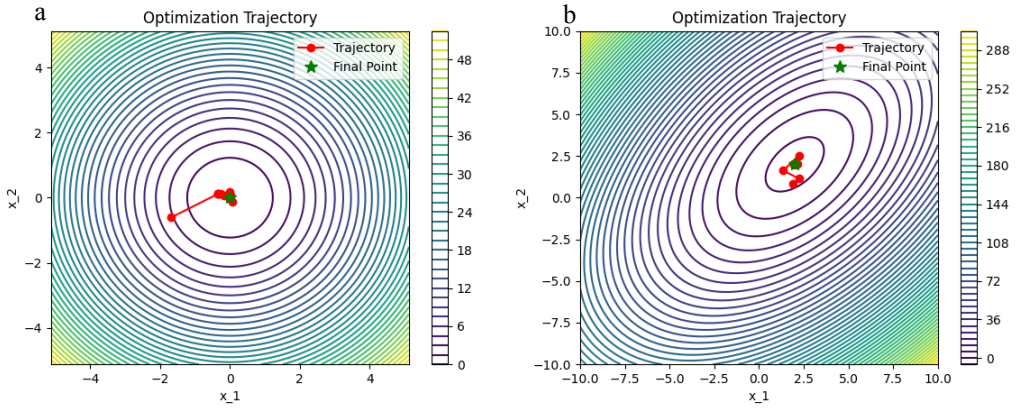

Рис. 2: Траектория поиска для проблем .... a - `Sphere`, b - `Trid`

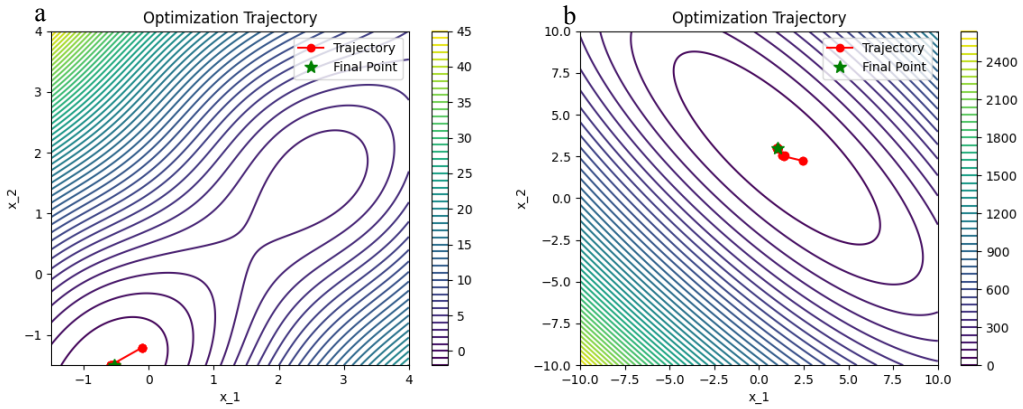

Рис. 3: Траектория поиска для проблем .... a - `McCormick`, b - `Booth`

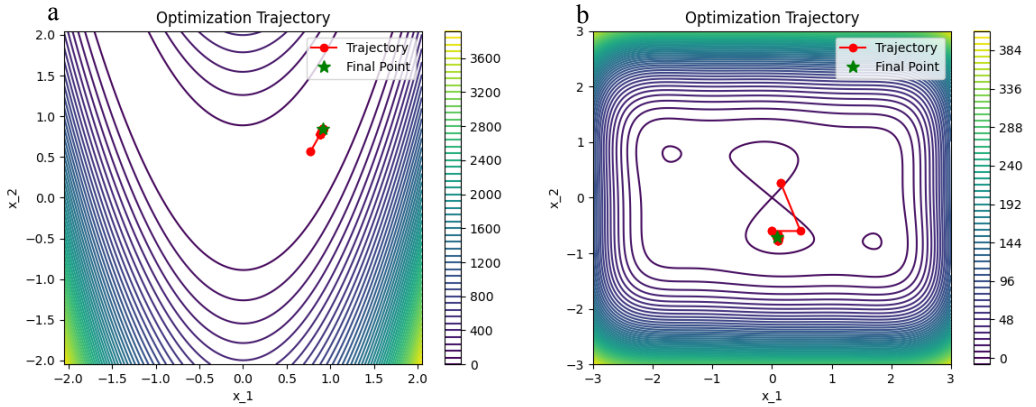

Рис. 4: Траектория поиска для проблем.... a - `Rosenbrock`, b - `Six-Hump Camel`

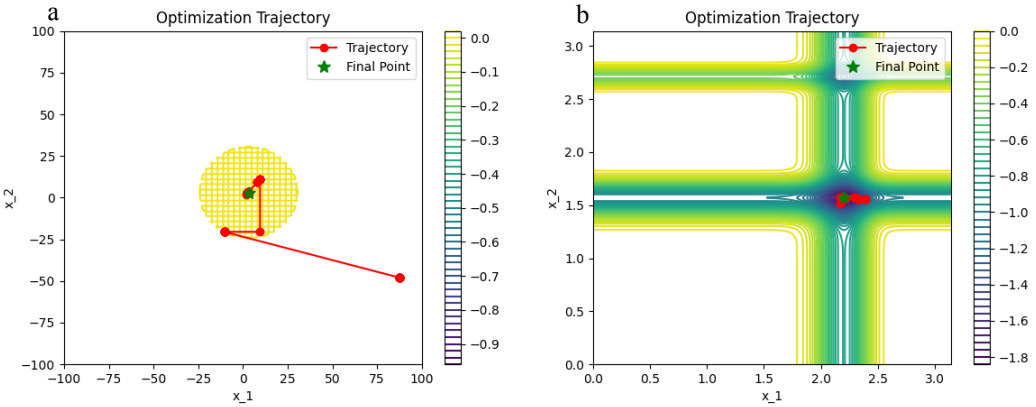

Рис. 5: Траектория поиска для проблем.... a - `Easom`, b - `Michalwicz`

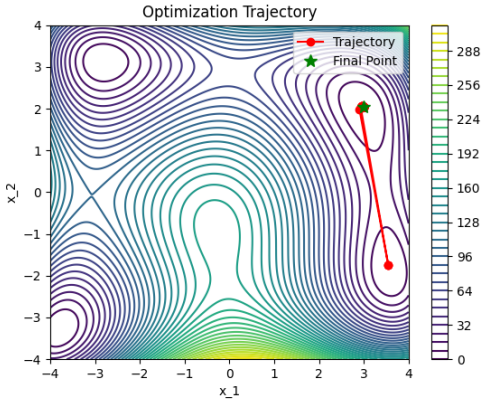

Рис. 6: Траектория поиска для проблемы Himmelblau

,

4. Выводы

Таблица 1: Результаты работы алгоритма на различных тестовых функциях

Тестовая функция	**Количество вызовов целевой функции**	x_{min}	f_{min}
Ackley	7588	$[8.52 \cdot 10^{-15}; -7.1 \cdot 10^{-15}]$	$3.19 \cdot 10^{-14}$
Eggholder	6704	$[439; 454]$	-935
Sphere	7446	$[-2.68 \cdot 10^{-13}; 2.09 \cdot 10^{-14}]$	$7.23 \cdot 10^{-26}$
Trid	6668	$[1.9(9); 1.9(9)]$	$-1.9(9)$
McCormick	6426	$[-0.52; -1.5]$	-1.91
Booth	6807	$[0.99; 3]$	$1.33 \cdot 10^{-5}$
Rosenbrock	7577	$[0.94; 0.90]$	0.002
Six-Hump Camel	6541	$[0.09; -0.71]$	-1.031
Easom	6541	$[3.14; 3.14]$	$-0.9(9)$
Michalewicz	6541	$[2.2; 1.57]$	-1.801
Himmelblau	6787	$[2.99; 2.04]$	0.033

Приложение А. Исходный код

```
import numpy as np
import matplotlib.pyplot as plt


def EVopt(f, dim, n_particles, max_iter, bounds):
    """
    Evolutionary Optimization Function.

    Parameters:
        f (function): Objective function to minimize.
        dim (int): Dimension of the search space.
        n_particles (int): Number of particles.
        max_iter (int): Maximum number of iterations.
        bounds (tuple): Tuple containing the lower and upper bounds of the search space.

    Returns:
        tuple: xmin (best position), fmin (minimum value), neval (function evaluations), coords
        ↪  (trajectory).
    """
    var_min = np.full(dim, bounds[0])
    var_max = np.full(dim, bounds[1])

    # Initialization
    particles = np.random.uniform(var_min, var_max, (n_particles, dim))
    nels = np.array([f(p) for p in particles])
    neval = n_particles  # Function evaluations

    best_idx = np.argmin(nels)
    best_nel = nels[best_idx]
    worst_nel = nels.max()
    xmin = particles[best_idx].copy()

    coords = np.zeros((max_iter, dim))
    coords[0] = xmin

    # Main Loop
    for iteration in range(max_iter):
        xnew, xnew1, xnew2 = [], [], []

        for i in range(n_particles):
            enrichment_bound = nels.mean()
            neli = nels[i]

            if neli > enrichment_bound:
                stability_bound = np.random.rand()
                stability_level = (neli - best_nel) / (worst_nel - best_nel)

                if stability_level > stability_bound:
                    alpha_index1 = np.random.randint(dim)
                    alpha_index2 = np.random.choice(dim, alpha_index1, replace=False)

                    xnew1.append(particles[i].copy())
                    xnew1[-1][alpha_index2] = xmin[alpha_index2]

                    gamma_index1 = np.random.randint(dim)
                    gamma_index2 = np.random.choice(dim, gamma_index1, replace=False)
                    dist = np.linalg.norm(particles - particles[i], axis=1)

                    nearest_idx = np.argsort(dist)[1] if len(dist) > 1 else None
                    if nearest_idx is not None:
                        xng = particles[nearest_idx].copy()
                        xnew2.append(particles[i].copy())
                        xnew2[-1][gamma_index2] = xng[gamma_index2]
                else:
                    xcp = particles.mean(axis=0)
                    xnew1.append(
                     particles[i] + (np.random.rand() * xmin - np.random.rand() * xcp) / stability_level
                    )

                    dist = np.linalg.norm(particles - particles[i], axis=1)
                    nearest_idx = np.argsort(dist)[1] if len(dist) > 1 else None
                    if nearest_idx is not None:
                        xng = particles[nearest_idx].copy()
                        xnew2.append(particles[i] + (np.random.rand() * xmin - np.random.rand() * xng))

            else:
                xnew.append(particles[i] + np.random.rand())

        # Clip to bounds and concatenate
        xnew = np.clip(np.array(xnew), var_min, var_max) if xnew else np.empty((0, dim))
        xnew1 = np.clip(np.array(xnew1), var_min, var_max) if xnew1 else np.empty((0, dim))
        xnew2 = np.clip(np.array(xnew2), var_min, var_max) if xnew2 else np.empty((0, dim))

        particles = np.vstack([particles, xnew, xnew1, xnew2])
        nels = np.array([f(p) for p in particles])
        neval += len(particles)
```

```
        # Select the best particles
        order = np.argsort(nels)
        particles = particles[order][:n_particles]
        nels = nels[order][:n_particles]

        best_nel = nels.min()
        worst_nel = nels.max()
        xmin = particles[np.argmin(nels)].copy()

        coords[iteration] = xmin

    fmin = best_nel
    return xmin, fmin, neval, coords


def testSuit(function_name):
    functions = {
        # Many local minima
        "Ackley": (lambda x: -20 * np.exp(-0.2 * np.sqrt(0.5 * (x[0] ** 2 + x[1] ** 2))) -
                             np.exp(0.5 * (np.cos(2 * np.pi * x[0]) + np.cos(2 * np.pi * x[1]))) + np.e +
                             ↪  20,
                   2, [-5, 5], [0, 0], 0),
       "Eggholder": (lambda x: -(x[1] + 47) * np.sin(np.sqrt(abs(x[1] + x[0] / 2 + 47))) - x[0] * np.sin(
            np.sqrt(abs(x[0] - (x[1] + 47)))),
                      2, [-512, 512], [512, 404.2319], -959.6407),
        # Bowl-shaped
        "Sphere": (lambda x: sum(xi ** 2 for xi in x), 2, [-5.12, 5.12], [0, 0], 0),
        "Trid": (lambda x: sum((xi - 1) ** 2 for xi in x) - sum(x[i] * x[i - 1] for i in range(1,
        ↪  len(x))),
                 2, [-4, 4], [2, 2], -2),
        # Plate-shaped
        "McCormick": (lambda x: np.sin(x[0] + x[1]) + (x[0] - x[1]) ** 2 - 1.5 * x[0] + 2.5 * x[1] + 1,
                      2, [-1.5, 4], [-0.54719, -1.54719], -1.9133),
        "Booth": (lambda x: (x[0] + 2 * x[1] - 7) ** 2 + (2 * x[0] + x[1] - 5) ** 2,
                  2, [-10, 10], [1, 3], 0),
        # Valley-shaped
        "Rosenbrock": (lambda x: sum(100 * (x[i + 1] - x[i] ** 2) ** 2 + (x[i] - 1) ** 2 for i in
        ↪  range(len(x) - 1)),
                       2, [-2.048, 2.048], [1, 1], 0),
        "Six-Hump Camel": (
        lambda x: (4 - 2.1 * x[0] ** 2 + (x[0] ** 4) / 3) * x[0] ** 2 + x[0] * x[1] + (-4 + 4 * x[1] **
        ↪  2) * x[1] ** 2,
        2, [-3, 3], [0.0898, -0.7126], -1.0316),
        # Steep ridges/drops
        "Michalewicz": (
       lambda x: -sum(np.sin(x[i]) * np.sin(((i + 1) * x[i] ** 2) / np.pi) ** 20 for i in range(len(x))),
        2, [0, np.pi], [2.20, 1.57], -1.8013),
       "Easom": (lambda x: -np.cos(x[0]) * np.cos(x[1]) * np.exp(-((x[0] - np.pi) ** 2 + (x[1] - np.pi)
        ↪  ** 2)),
                  2, [-100, 100], [np.pi, np.pi], -1),
       "Himmelblau": (lambda x: (x[0] ** 2 + x[1] - 11) ** 2 + (x[0] + x[1] ** 2 - 7) ** 2, 2, [-4, 4],
        ↪  [3, 2], 0)
    }
    return functions[function_name]


def plot_trajectory(f, bounds, coords):
    x = np.linspace(bounds[0], bounds[1], 500)
    y = np.linspace(bounds[0], bounds[1], 500)
    X, Y = np.meshgrid(x, y)
    Z = np.array([[f([xi, yi]) for xi, yi in zip(x_row, y_row)] for x_row, y_row in zip(X, Y)])

    plt.figure()
    plt.contour(X, Y, Z, levels=50, cmap='viridis')
    plt.colorbar()
    plt.xlabel('x_1')
    plt.ylabel('x_2')
    plt.title('Optimization Trajectory')

    plt.plot(coords[:, 0], coords[:, 1], 'r-o', label='Trajectory')
    plt.plot(coords[-1, 0], coords[-1, 1], 'g*', label='Final Point', markersize=10)
    plt.legend()
    plt.show()


def main():
    function_name = "Himmelblau"  # Change to desired function
    f, dim, bounds, xmin_true, fmin_true = testSuit(function_name)

    n_particles = 30
    max_iter = 100

    xmin, fmin, neval, coords = EVopt(f, dim, n_particles, max_iter, bounds)
    plot_trajectory(f, bounds, coords)

    print(f"Test function: {function_name}")
    print(f"True minimum: x = {xmin_true}, f(x) = {fmin_true}")
    print(f"Obtained minimum: x = {xmin}, f(x) = {fmin}")
    print(f"Error in x: {np.linalg.norm(np.array(xmin) - np.array(xmin_true))}")
    print(f"Error in f(x): {abs(fmin - fmin_true)}")
```

```
    print(f"Number of Func Evaluations: {neval}")

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

Приложение Б. Блок-схема алгоритма

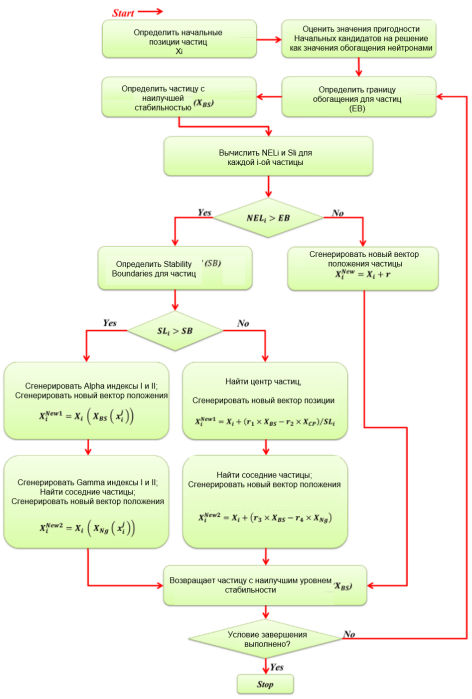

Рис. 7: Блок схема работы алгоритма

,