Санкт-Петербургский государственный университет

Факультет прикладной математики – процессов управления

**Отчет**

по лабораторной работе №5

по дисциплине «Алгоритмы и структуры данных»

на тему «Алгоритм роя частиц»

Вариант 4

Автор работы: Добренкова Л.С.

Группа: 22.Б15-пу

Преподаватель: Дик А.Г.

Санкт-Петербург, 2023

**Оглавление**

[**1.** **Введение** 3](#_Toc154463442)

[**2.** **Цель работы** 3](#_Toc154463443)

[**3.** **Теоретическая часть** 3](#_Toc154463444)

[**4.** **Модификация алгоритма** 4](#_Toc154463445)

[**5.** **Представление и спецификация программной части** 5](#_Toc154463446)

[**6.** **Контрольный пример** 6](#_Toc154463447)

[**7.** **Тестирование и анализ результатов работы алгоритма** 8](#_Toc154463448)

[**8.** **Сравнение с генетическим алгоритмом** 9](#_Toc154463449)

[**9.** **Выводы** 10](#_Toc154463450)

[**10.** **Список литературы** 10](#_Toc154463451)

[**11.** **Листинг** 10](#_Toc154463452)

# **Введение**

Алгоритм роя частиц (PSO), разработанная Кеннеди и Эберхартом, представляет собой относительно новый метод оптимизации, вдохновленный коллективным интеллектом. Как и другие эволюционные алгоритмы (EAs), PSO также является стохастическим алгоритмом поиска на основе популяции, однако в нем отсутствуют операторы кроссовера и мутации. В процессе поиска каждая частица корректирует свое поведение по опыту поиска своей предыдущей лучшей позиции и глобальной лучшей позиции. Благодаря своей простоте и легкости в реализации, PSO успешно применяется к различным практическим задачам оптимизации

# **Цель работы**

Исследование особенностей алгоритмов роевого интеллекта для решения задач глобальной оптимизации и сравнение с генетическим алгоритмом.

# **Теоретическая часть**

Метод роя частиц (PSO) является оптимизационным алгоритмом. В рамках PSO, каждая "частица" в пространстве поиска представляет собой потенциальное решение задачи оптимизации. Частицы перемещаются по пространству с целью минимизации (или максимизации) функции приспособленности.

Ниже приведен список основных понятий:

* **Частица (Particle):** Потенциальное решение задачи оптимизации, представленное в пространстве параметров.
* **Функция приспособленности (Fitness Function):** Оценочная функция, измеряющая качество решения в данной точке пространства параметров.
* **Положение частицы (Position):** Текущая координата частицы в пространстве параметров.
* **Лучшее личное положение (pbest):** Запоминает лучшую позицию, на которой данная частица достигла наилучшей приспособленности.
* **Лучшее глобальное положение (gbest):** Запоминает лучшую позицию среди всех частиц в рое.
* **Скорость частицы (Velocity):** Вектор, определяющий изменение положения частицы на каждом шаге оптимизации.
* **Коэффициенты ускорения (Acceleration Coefficients):** Параметры, определяющие влияние лучшего личного и глобального положения (также известны как когнитивный и социальный коэффициенты) на обновление скорости частицы.
* **Итерации (Generations):** Итерации оптимизационного процесса, на каждой из которых происходит обновление положения частиц.
* **Нормальное распределение (Normal Distribution, N(.) ):** Распределение вероятностей, описывающее случайные величины, которые сгруппированы вокруг среднего значения.
* **Адаптивность (Adaptability):** Способность алгоритма или параметров алгоритма изменяться в зависимости от условий задачи.

Основные шаги алгоритма:

1. **Инициализация:** 
   * Инициализация частиц с случайными положениями и лучшими личными положениями.
2. **Основной цикл оптимизации:**
   * Вычисление значений функции приспособленности в текущем положении и обновление глобального лучшего положения
   * Обновление скорости частицы в соответствии с лучшим глобальным и лучшем личным положением, а также коэффициентами ускорения.
   * Обновление положения частицы в соответствии со скоростью
3. **Повторение:**
   * Повторение шага 2 заданное количество раз (генераций).
4. **Вывод результатов:**
   * Вывод результатов

# **Модификация алгоритма**

В данной работе реализована модификация BareBonesPSO, включающая в себя изменения в механизме обновления положения частиц. Вместо использования стандартных формул, содержащих коэффициенты ускорения, предложен подход, основанный на нормальном распределении значений скорости; частица более не имеет атрибута скорости, положение на t+1 итерации вычисляется по формуле:

,

*Рисунок 4.1 Формула вычисления положения частицы в зависимости от итерации*

Эта модификация предполагает более гибкое и адаптивное обновление положения частиц, что может привести к более эффективной оптимизации функций. Также, что немаловажно, эта модификация избавляет исследователя от необходимости “в ручную” подбирать параметры, такие как коэффициенты ускорения. Из незначительных плюсов можно выделить сравнительную простоту написания алгоритма – программная реализация алгоритма содержит менее 50 строк кода.

# **Представление и спецификация программной части**

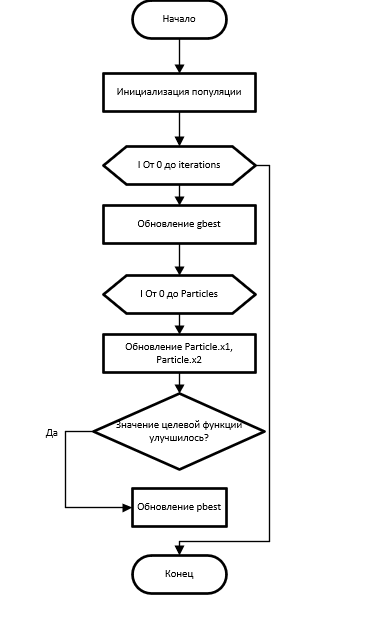
Программная реализация выполнена на языке Python3.11 с использованием библиотеки numpy для нахождения нормального распределения и библиотеки PyQT для визуализации алгоритма.

Нельзя не отметить, что программа выполнена в функциональном стиле и не содержит классов (за исключением GUI); вся логика программы заключена в следующих функциях:

1. **Particle** - TypedDict, представляющий структуру данных для частиц. Хоть и формально является классом, де-факто используется как аналог C++ struct в Python3
2. **initialize\_particles** - функция инициализации частиц.
3. **func**- функция, вычисляющая значение целевой функции .
4. **algorithm** - основная функция, реализующая алгоритм PSO.

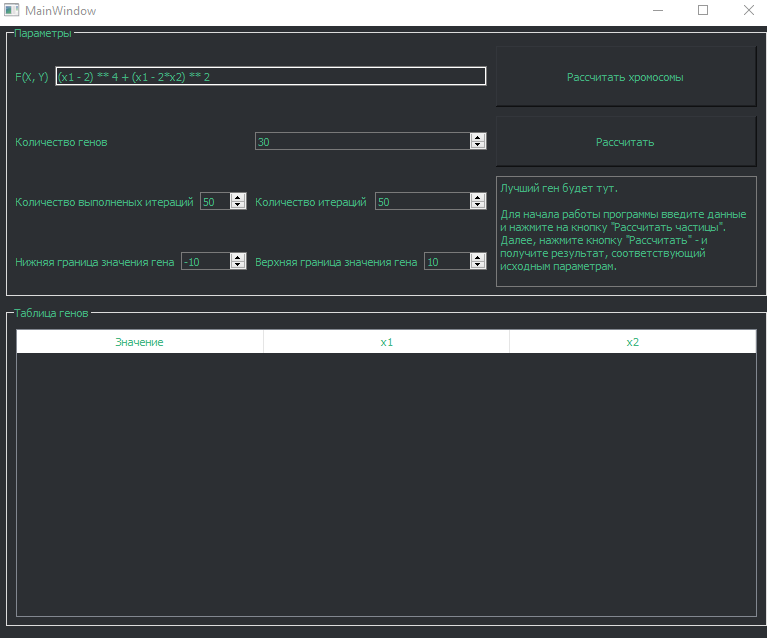
На рисунке ниже представлена блок-схема алгоритма:

*Блок-схема 5.1 Блок-схема алгоритма*

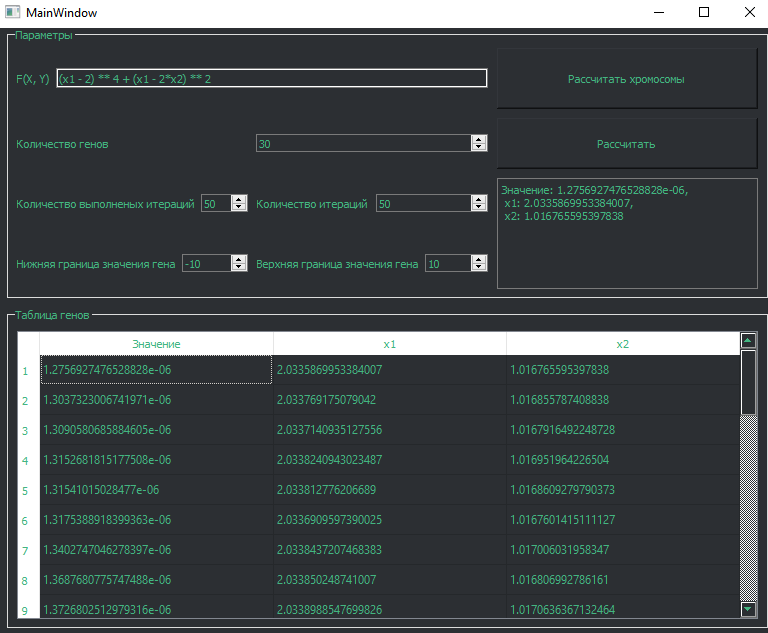


# **Контрольный пример**

При запуске программы пользователь увидит MainWindow с предустановленными параметрами. В одном из окон он может наблюдать инструкцию по использованию программы.



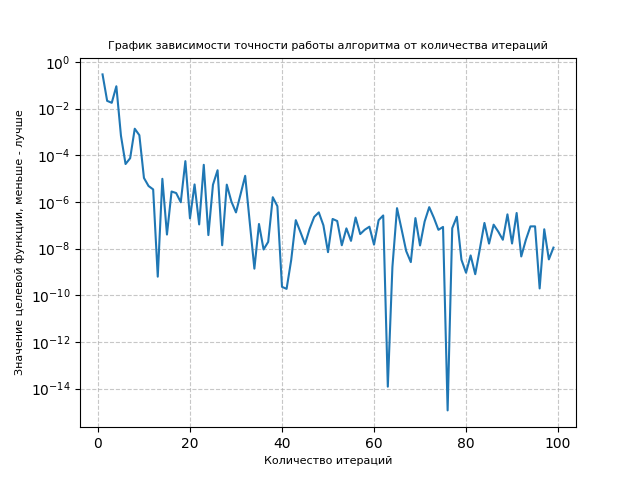
*Рисунок 6.1 Main Window непосредственно при запуске программы*

**

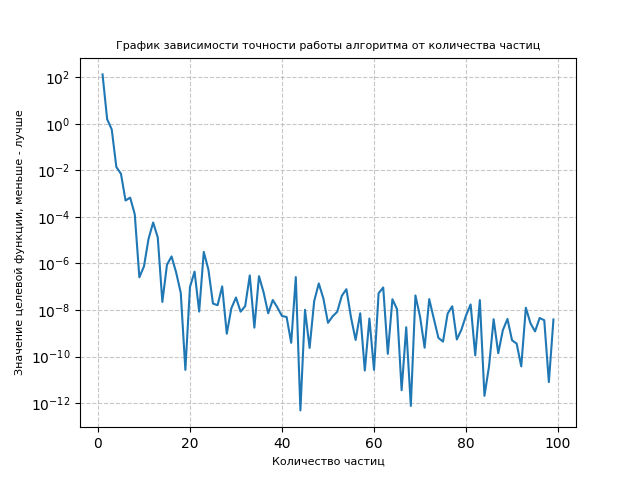
*Рисунок 6.2 Результат работы программы*

# **Тестирование и анализ результатов работы алгоритма**

На рисунках 7.1-7.2 представлены результаты тестирования программы. Начальные данные: область [0, 10], функция  с минимумом, равным 0, в точке (2, 1). Количество генов в графике 7.1 – 30, количество итераций во втором графике – 50.



*Рисунок 7.1 График зависимости  
 точности результатов алгоритма от количества итераций.*

**

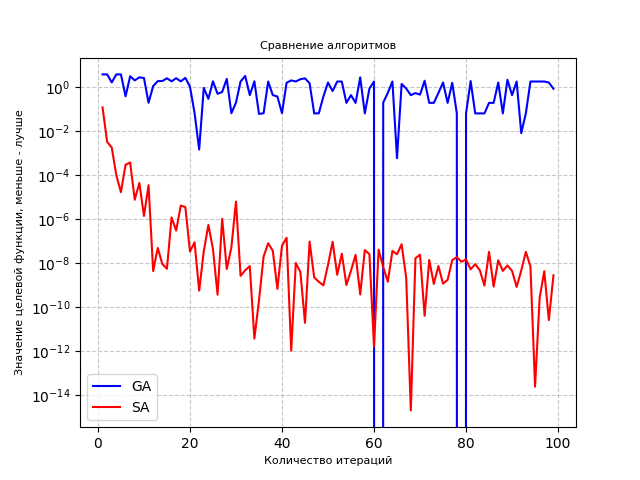
*Рисунок 7.2 График зависимости  
 точности результатов алгоритма от количества частиц.*

Как видно из рисунков 7.1-7.2, увеличение количества частиц и итераций не гарантирует повышения точности из-за стохастической природы PSO; при определенном везении даже при относительно малых значениях параметров возможно получение результата с более высокой точностью (см. впадины на графиках).

Тем не менее, при совокупном увеличении параметров алгоритм начинает работать более стабильно, и вероятность просадок в точности значительно уменьшается.

Автор рекомендует выставить количество итераций и частиц равным 50 и 50 соответственно для получения оптимального соотношения точности ко времени.

# **Сравнение с генетическим алгоритмом**

На графике 8.1 представлено сравнение с генетическим алгоритмом (лабораторная работа 4). 

*Рисунок 7.2 График зависимости  
 точности результатов ГА(GA) и PSO(SA) в зависимости от итераций*

Как видно из графика, генетический алгоритм проигрывает в средней точности роевому алгоритму, но иногда, если повезет, может сойтись весьма точно. Выбросы (впадины синей линии), как было показано на графиках в предыдущей работе, могут происходить практически вне зависимости от параметров, поэтому ГА может быть полезнее там, где требуется примерный и быстрый ответ, а SA – там где важна воспроизводимость ответа и точность с trade-off’ом по времени.

# **Выводы**

Были получены все нужные навыки для реализации роевых алгоритмов, а также реализован алгоритм BareBonesPSO, а также интерфейс к нему. Проанализированы результаты работы, выполнено сравнение с Генетическим алгоритмом.

# **Список литературы**

[1] Статья про BareBonesPSO <https://www.hindawi.com/journals/mpe/2013/175848/>

# **Листинг**

# Importing libraries and modules  
import random  
import struct  
from copy import deepcopy  
from typing import TypedDict  
import numpy as np  
import matplotlib.pyplot as plt  
  
# Definition of TypedDict for the particle structure  
class Particle(TypedDict):  
 x1: float  
 x2: float  
 pbest\_x1: float  
 pbest\_x2: float  
  
# Initializing particles with random values within specified bounds  
def initialize\_particles(bounds, amount):  
 *"""  
 Initialize particles with random values.  
  
 Parameters:  
 - bounds: Range of values for initialization (list, [min, max]).  
 - amount: Number of particles.  
  
 Returns:  
 - List of particles in TypedDict format.  
 """* particles = []  
 for \_ in range(amount):  
 particles.append(Particle(  
 x1=random.uniform(bounds[0], bounds[1]),  
 x2=random.uniform(bounds[0], bounds[1]),  
 pbest\_x1=random.uniform(0, 1),  
 pbest\_x2=random.uniform(0, 1)  
 ))  
 return particles  
  
# Definition of the objective function  
def func(f, x1, x2):  
 *"""  
 Compute the value of the objective function.  
  
 Parameters:  
 - f: String with the mathematical expression of the objective function.  
 - x1, x2: Variable values.  
  
 Returns:  
 - Value of the objective function for the given variables.  
 """* return eval(f)  
  
# Implementation of the PSO-based optimization algorithm  
def algorithm(particles, generations, f, bounds):  
 *"""  
 Implementation of the PSO-based optimization algorithm.  
  
 Parameters:  
 - particles: Initial population of particles.  
 - generations: Number of generations (iterations) of the algorithm.  
 - f: String with the mathematical expression of the objective function.  
 - bounds: Range of values for initialization (list, [min, max]).  
  
 Returns:  
 - Evolution history of the population at each generation.  
 """* history = [deepcopy(sorted(particles, key=lambda x: func(f, x['x1'], x['x2'])))]  
 for \_ in range(generations):  
 gbest = (history[-1][0]['x1'], history[-1][0]['x2'])  
 for particle in particles:  
 f\_was = func(f, particle['x1'], particle['x2'])  
 particle['x1'] = np.random.normal((gbest[0] + particle['pbest\_x1']) / 2,  
 abs(gbest[0] - particle['pbest\_x1']))  
 particle['x2'] = np.random.normal((gbest[1] + particle['pbest\_x2']) / 2,  
 abs(gbest[1] - particle['pbest\_x2']))  
 f\_now = func(f, particle['x1'], particle['x2'])  
 if f\_now < f\_was:  
 particle['pbest\_x1'] = particle['x1']  
 particle['pbest\_x2'] = particle['x2']  
 particles = sorted(particles, key=lambda x: func(f, x['x1'], x['x2']))  
 history.append(deepcopy(particles))  
 return history