Санкт-Петербургский государственный университет

Факультет прикладной математики-процессов управления

Кафедра компьютерного моделирования и многопоточных систем

Лабораторная работа по дисциплине
«Алгоритмы и структуры данных»

«Разработка и реализация алгоритма роевого интеллекта для решения задач глобальной оптимизации»

Выполнил:
Зайнуллин Мансур Альбертович
Группа: 23.Б16-пу

Руководитель:
Дик Александр Генадьевич
ассистент кафедры компьютерного моделирования
и многопоточных систем

Санкт-Петербург
2024

Оглавление

1 Цель работы

Цель работы — исследование особенностей алгоритмов роевого интеллекта для решения задач глобальной оптимизации и сравнение с генетическим алгоритмом.

2 Описание алгоритма

Алгоритм роя частиц (Particle Swarm Optimization, PSO) — это метод оптимизации, вдохновлённый коллективным поведением в природе, таким как стаи птиц или рои насекомых. Он используется для поиска оптимальных решений в многомерных пространствах.

2.1 Основные компоненты алгоритма

1. **Инициализация**:
    - Каждая частица в рое представляет потенциальное решение задачи.
    - Частицы инициализируются случайными позициями и скоростями в пределах допустимого пространства поиска.
2. **Обновление скорости**:
    - Скорость каждой частицы обновляется на основе трёх компонентов:
        - **Инерционная составляющая**: сохраняет текущую траекторию частицы, помогая ей двигаться в том же направлении.
        - **Когнитивная составляющая**: направляет частицу к её лучшей найденной позиции, стимулируя индивидуальное исследование.
        - **Социальная составляющая**: направляет частицу к лучшей позиции, найденной всем роем, способствуя коллективному обучению.
    - Формула обновления скорости:

$$v_i(t+1) = w \cdot v_i(t) + c_1 \cdot r_1 \cdot (p_i - x_i(t)) + c_2 \cdot r_2 \cdot (g - x_i(t))$$

    где:

- w — коэффициент инерции;
- c_1 и c_2 — коэффициенты когнитивного и социального влияния;
- r_1 и r_2 — случайные числа в интервале [0, 1];
- p_i — лучшая позиция частицы;
- g — лучшая позиция роя.

3. **Обновление позиции**:

   - Позиция частицы обновляется на основе её текущей скорости:

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

4. **Ограничение скорости**:

   - Для предотвращения слишком больших изменений, скорость часто ограничивается максимальным значением $V_{\max}$.

5. **Оценка и обновление**:

   - Каждая частица оценивается с помощью функции приспособленности.
   - Обновляются лучшие личные и глобальные позиции.

2.2 Цель алгоритма

Алгоритм роя частиц стремится найти глобальный минимум (или максимум) целевой функции, эффективно исследуя пространство решений и избегая локальных минимумов.

3 Описание схемы пошагового выполнения алгоритма и блок-схемы

3.1 Пошаговое описание алгоритма

Алгоритм роя частиц (PSO) выполняется следующим образом:

1. **Инициализация**:
   - Каждая частица получает случайную начальную позицию и скорость в пределах заданного пространства поиска.
2. **Обновление скорости**:
   - Скорость каждой частицы обновляется с учётом инерционной, когнитивной и социальной составляющих:

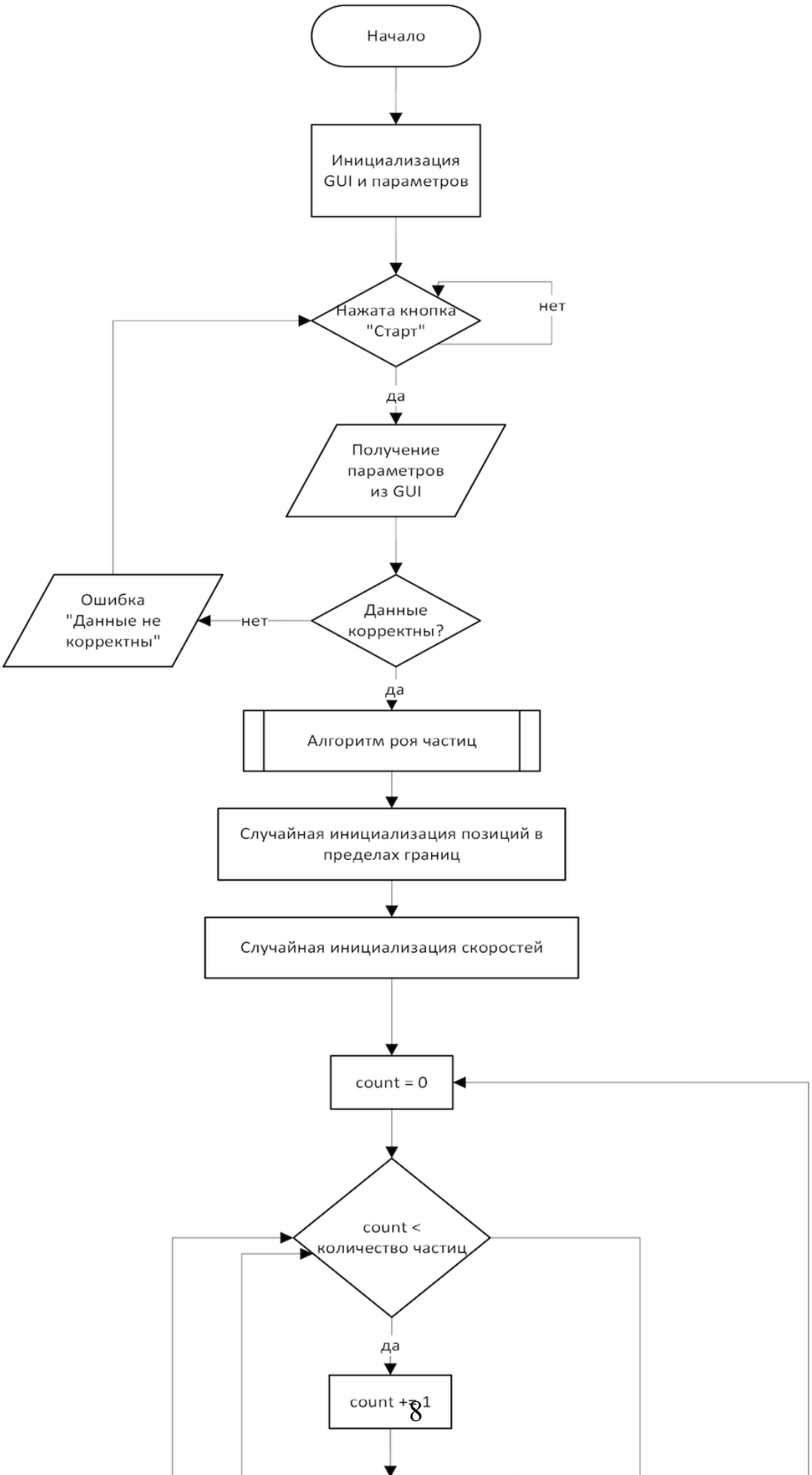
$$v_i(t+1) = w \cdot v_i(t) + c_1 \cdot r_1 \cdot (p_i - x_i(t)) + c_2 \cdot r_2 \cdot (g - x_i(t))$$

3. **Ограничение скорости**:
   - Применяется ограничение на максимальную скорость, чтобы предотвратить слишком большие изменения в позициях частиц.
4. **Обновление позиции**:
   - Позиция каждой частицы обновляется на основе её текущей скорости:

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

5. **Оценка и обновление**:
   - Оценивается приспособленность текущих позиций частиц.
   - Обновляются лучшие личные и глобальные позиции.
6. **Проверка условий завершения**:
   - Алгоритм проверяет, достигнуто ли заданное число итераций или приемлемый уровень точности.

3.2 Блок-схемы



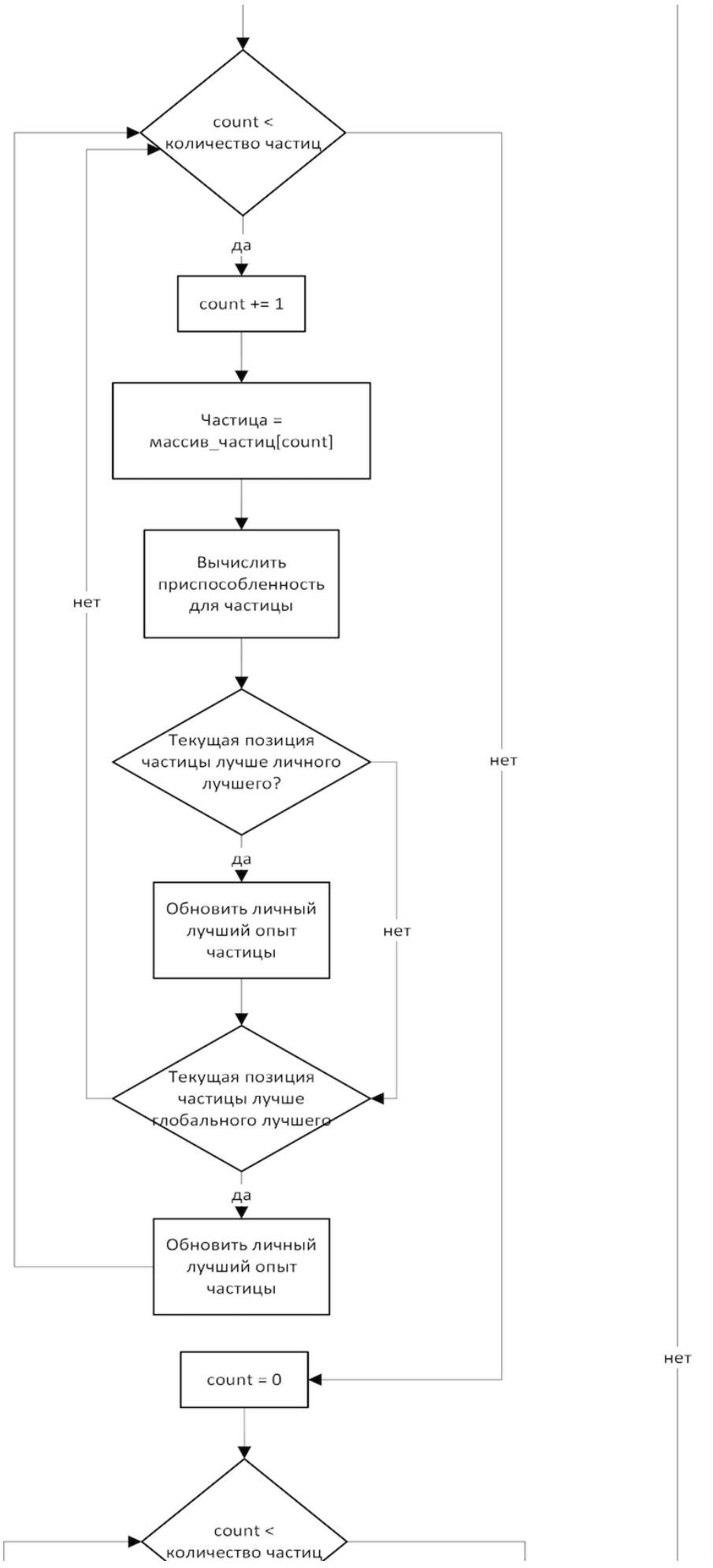


Рис. 2 Блок-схема 2

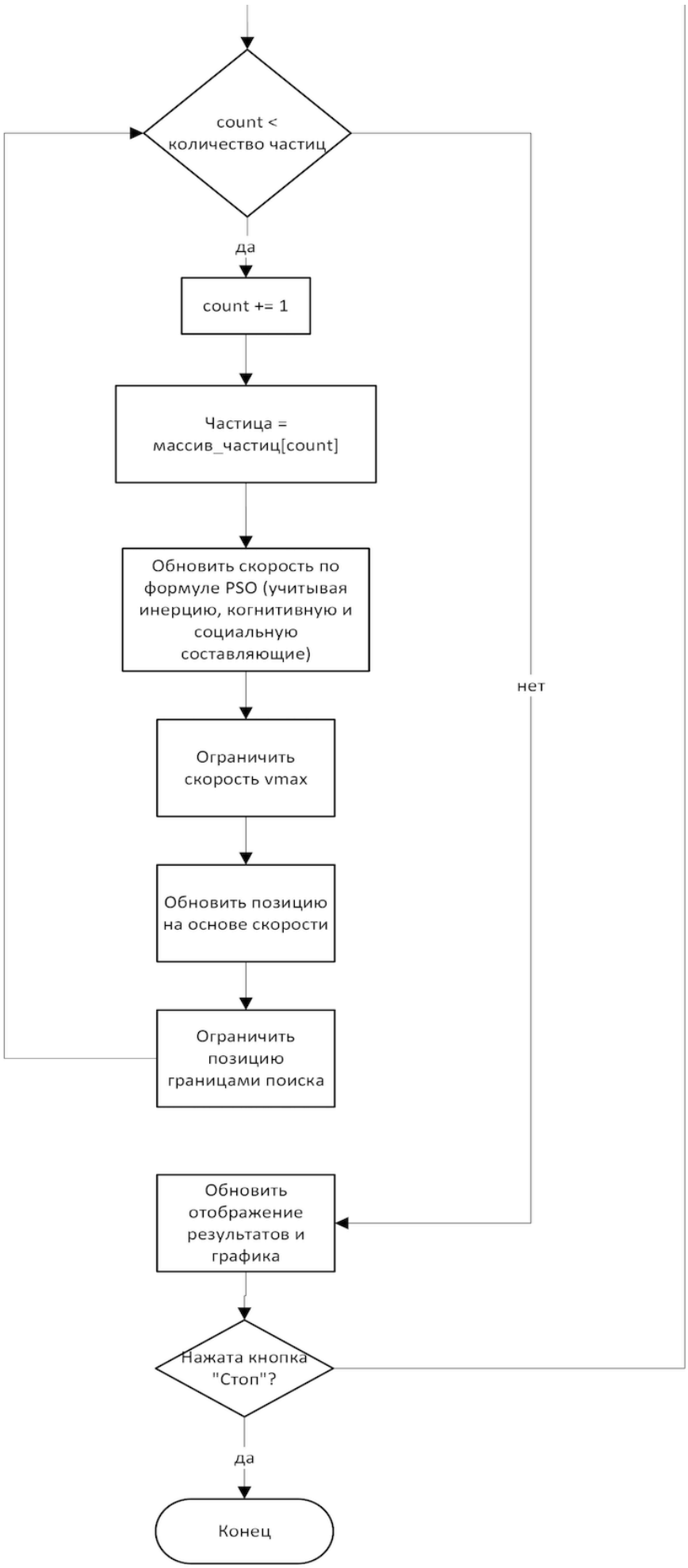


Рис. 3 Блок-схема 3

4 Формализация задачи

Алгоритм роя частиц (Particle Swarm Optimization, PSO) использует следующие основные формулы для обновления скоростей и позиций частиц в пространстве поиска:

Обновление скорости:

$$v_i(t+1) = w \cdot v_i(t) + c_1 \cdot r_1 \cdot (p_i - x_i(t)) + c_2 \cdot r_2 \cdot (g - x_i(t))$$

где:

- w — коэффициент инерции;
- c_1 и c_2 — когнитивный и социальный коэффициенты;
- r_1 и r_2 — случайные числа в интервале [0, 1];
- p_i — лучшая позиция частицы;
- g — лучшая позиция роя.

Ограничение скорости:

$$v_{ij}(t+1) = \begin{cases} v_{ij}(t+1), & \text{если } v_{ij}(t+1) \leq V_{\max,j}, \\ V_{\max,j}, & \text{если } v_{ij}(t+1) > V_{\max,j}. \end{cases}$$

Обновление позиции:

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

Эти формулы описывают, как частицы перемещаются в пространстве поиска, стремясь найти оптимальное решение.

4.1 Спецификация программы

Table 1: Спецификация функций программы

Имя функции	**Тип возвращаемого значения**	**Описание функции**
__init__	None	Инициализация частицы с заданными границами и максимальной скоростью.
objective_function	float	Вычисляет значение целевой функции для заданных координат x и y.
update_velocity	None	Обновляет скорость частицы на основе инерционной, когнитивной и социальной составляющих.
optimize	None	Основной цикл оптимизации, обновляющий позиции и скорости частиц.
update_plot	None	Обновляет графическое отображение текущих позиций частиц.
update_best_result	None	Обновляет отображение лучших найденных позиций и значений функции.
create_widgets	None	Создает и размещает виджеты интерфейса пользователя.
start_optimization	None	Запускает процесс оптимизации, инициализируя частицы и параметры.
stop_optimization	None	Останавливает процесс оптимизации.
reset_optimization	None	Сбрасывает состояние алгоритма и графика для начала новой оптимизации.

5 Листининг

```
import tkinter as tk
from tkinter import messagebox
import threading
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

class Particle:
    def __init__(self, bounds, vmax):
        self.position = np.array([np.random.uniform(low, high) for low,
            ↪ high in bounds])
        self.velocity = np.random.uniform(-vmax, vmax, len(bounds))
        self.best_position = self.position.copy()
        self.best_fitness = float('inf')

def objective_function(x, y):
    return 8 * (x ** 2) + 4 * x * y + 5 * (y ** 2)

def update_velocity(particle, global_best_position, w, c1, c2, vmax,
    ↪ use_velocity_clamping):
    r1 = np.random.rand(len(particle.position))
    r2 = np.random.rand(len(particle.position))
    cognitive = c1 * r1 * (particle.best_position - particle.position)
    social = c2 * r2 * (global_best_position - particle.position)
    particle.velocity = w * particle.velocity + cognitive + social
    if use_velocity_clamping:
        particle.velocity = np.clip(particle.velocity, -vmax, vmax)

class PSOApp:
    def __init__(self, master):
        self.master = master
        self.master.title("          ")
        self.master.configure(bg='#FFFACD')

        self.is_running = False
        self.should_reset = False
```

```
        self.create_widgets()

        self.particles = []
        self.global_best_position = None
        self.global_best_fitness = float('inf')
        self.bounds = []

    def create_widgets(self):
        param_frame = tk.Frame(self.master, bg='#FFFACD')
        param_frame.pack(side=tk.TOP, fill=tk.X, padx=10, pady=10)

        tk.Label(param_frame, text="          :", bg='#FFFACD').grid(row=0,
            ↪ column=0, sticky=tk.W)
        self.population_size_entry = tk.Entry(param_frame)
        self.population_size_entry.insert(0, "30")
        self.population_size_entry.grid(row=0, column=1)

        tk.Label(param_frame, text="           (w):",
            ↪ bg='#FFFACD').grid(row=1, column=0, sticky=tk.W)
        self.w_entry = tk.Entry(param_frame)
        self.w_entry.insert(0, "0.5")
        self.w_entry.grid(row=1, column=1)

        tk.Label(param_frame, text="             (c1):",
            ↪ bg='#FFFACD').grid(row=2, column=0, sticky=tk.W)
        self.c1_entry = tk.Entry(param_frame)
        self.c1_entry.insert(0, "1.0")
        self.c1_entry.grid(row=2, column=1)

        tk.Label(param_frame, text="             (c2):",
            ↪ bg='#FFFACD').grid(row=3, column=0, sticky=tk.W)
        self.c2_entry = tk.Entry(param_frame)
        self.c2_entry.insert(0, "1.0")
        self.c2_entry.grid(row=3, column=1)

        tk.Label(param_frame, text="            (vmax):",
            ↪ bg='#FFFACD').grid(row=4, column=0, sticky=tk.W)
        self.vmax_entry = tk.Entry(param_frame)
        self.vmax_entry.insert(0, "2.0")
        self.vmax_entry.grid(row=4, column=1)
```

```
tk.Label(param_frame, text="X Min:", bg='#FFFACD').grid(row=5,
    ↪ column=0, sticky=tk.W)
self.x_min_entry = tk.Entry(param_frame)
self.x_min_entry.insert(0, "-10")
self.x_min_entry.grid(row=5, column=1)

tk.Label(param_frame, text="X Max:", bg='#FFFACD').grid(row=5,
    ↪ column=2, sticky=tk.W)
self.x_max_entry = tk.Entry(param_frame)
self.x_max_entry.insert(0, "10")
self.x_max_entry.grid(row=5, column=3)

tk.Label(param_frame, text="Y Min:", bg='#FFFACD').grid(row=6,
    ↪ column=0, sticky=tk.W)
self.y_min_entry = tk.Entry(param_frame)
self.y_min_entry.insert(0, "-10")
self.y_min_entry.grid(row=6, column=1)

tk.Label(param_frame, text="Y Max:", bg='#FFFACD').grid(row=6,
    ↪ column=2, sticky=tk.W)
self.y_max_entry = tk.Entry(param_frame)
self.y_max_entry.insert(0, "10")
self.y_max_entry.grid(row=6, column=3)

#                   /
self.velocity_clamping_var = tk.BooleanVar(value=True)
self.velocity_clamping_check = tk.Checkbutton(param_frame,
    ↪ text="            ", variable=self.velocity_clamping_var,
    ↪ bg='#FFFACD')
self.velocity_clamping_check.grid(row=7, column=0, columnspan=2,
    ↪ sticky=tk.W)

button_frame = tk.Frame(self.master, bg='#FFFACD')
button_frame.pack(side=tk.TOP, pady=10)

self.start_button = tk.Button(button_frame, text="   ",
    ↪ command=self.start_optimization)
self.start_button.pack(side=tk.LEFT, padx=5)
```

```
        self.stop_button = tk.Button(button_frame, text="  ",
            ↪ command=self.stop_optimization)
        self.stop_button.pack(side=tk.LEFT, padx=5)

        self.reset_button = tk.Button(button_frame, text="   ",
            ↪ command=self.reset_optimization)
        self.reset_button.pack(side=tk.LEFT, padx=5)

        plot_frame = tk.Frame(self.master, bg='#FFFACD')
        plot_frame.pack(side=tk.TOP, fill=tk.BOTH, expand=True)

        self.figure = plt.Figure()
        self.ax = self.figure.add_subplot(1, 1, 1)

        self.canvas = FigureCanvasTkAgg(self.figure, master=plot_frame)
        self.canvas.get_tk_widget().pack(side=tk.TOP, fill=tk.BOTH,
            ↪ expand=True)

        result_frame = tk.Frame(self.master, bg='#FFFACD')
        result_frame.pack(side=tk.TOP, fill=tk.X, padx=10, pady=10)

        tk.Label(result_frame, text="        :", bg='#FFFACD').grid(row=0,
            ↪ column=0, sticky=tk.W)
        self.best_position_label = tk.Label(result_frame, text="",
            ↪ bg='#FFFACD')
        self.best_position_label.grid(row=0, column=1, sticky=tk.W)

        tk.Label(result_frame, text="            :",
            ↪ bg='#FFFACD').grid(row=1, column=0, sticky=tk.W)
        self.best_fitness_label = tk.Label(result_frame, text="",
            ↪ bg='#FFFACD')
        self.best_fitness_label.grid(row=1, column=1, sticky=tk.W)

    def start_optimization(self):
        if not self.is_running:
            try:
                population_size = int(self.population_size_entry.get())
                w = float(self.w_entry.get())
                c1 = float(self.c1_entry.get())
                c2 = float(self.c2_entry.get())
```

```
            vmax = float(self.vmax_entry.get())
            x_min = float(self.x_min_entry.get())
            x_max = float(self.x_max_entry.get())
            y_min = float(self.y_min_entry.get())
            y_max = float(self.y_max_entry.get())

            self.bounds = [(x_min, x_max), (y_min, y_max)]

            if self.should_reset or not self.particles:
                self.particles = [Particle(self.bounds, vmax) for _ in
                    ↪ range(population_size)]
                self.global_best_position = None
                self.global_best_fitness = float('inf')
                self.should_reset = False

            self.is_running = True

            use_velocity_clamping = self.velocity_clamping_var.get()
            threading.Thread(target=self.optimize, args=(w, c1, c2,
                ↪ vmax, use_velocity_clamping), daemon=True).start()
        except ValueError:
            messagebox.showerror("    ", "                .")

def stop_optimization(self):
    self.is_running = False

def reset_optimization(self):
    self.is_running = False
    self.should_reset = True
    self.particles = []
    self.global_best_position = None
    self.global_best_fitness = float('inf')
    self.best_position_label.config(text="")
    self.best_fitness_label.config(text="")
    self.ax.clear()
    self.canvas.draw()

def optimize(self, w, c1, c2, vmax, use_velocity_clamping):
    while self.is_running:
        for particle in self.particles:
```

```
            fitness = objective_function(particle.position[0],
                ↪ particle.position[1])
            if fitness < particle.best_fitness:
                particle.best_fitness = fitness
                particle.best_position = particle.position.copy()
            if fitness < self.global_best_fitness:
                self.global_best_fitness = fitness
                self.global_best_position = particle.position.copy()
        for particle in self.particles:
            update_velocity(particle, self.global_best_position, w, c1,
                ↪ c2, vmax, use_velocity_clamping)
            particle.position += particle.velocity
            for i in range(len(particle.position)):
                low, high = self.bounds[i]
                particle.position[i] = np.clip(particle.position[i], low,
                    ↪ high)
        self.update_plot()
        self.update_best_result()
        plt.pause(0.01)

def update_plot(self):
    self.ax.clear()
    x_vals = [particle.position[0] for particle in self.particles]
    y_vals = [particle.position[1] for particle in self.particles]
    self.ax.scatter(x_vals, y_vals, c='blue', label='    ')
    if self.global_best_position is not None:
        self.ax.scatter(self.global_best_position[0],
            ↪ self.global_best_position[1], c='red', marker='*', s=200,
            ↪ label='    ')
    self.ax.set_xlabel('X')
    self.ax.set_ylabel('Y')
    self.ax.legend()
    x_min, x_max = self.bounds[0]
    y_min, y_max = self.bounds[1]
    self.ax.set_xlim(x_min, x_max)
    self.ax.set_ylim(y_min, y_max)
    self.canvas.draw()

def update_best_result(self):
    if self.global_best_position is not None:
```

```
            pos_text = f"({self.global_best_position[0]:.4f},
                ↪ {self.global_best_position[1]:.4f})"
            self.best_position_label.config(text=pos_text)
            self.best_fitness_label.config(text=f"{self.global_best_fitness:.6f}")

if __name__ == "__main__":
    root = tk.Tk()
    app = PSOApp(root)
    root.mainloop()
```

6 Контрольный пример и результаты тестирования

6.1 Описание контрольного примера

Целью контрольного примера является оценка эффективности алгоритма роя частиц. В тестировании использовался размер популяции 100, диапазон значений переменных от -10 до 10.

6.2 Установка и настройка окружения

Для выполнения тестирования необходимо установить **Python 3.9** и **Git**. Клонируйте репозиторий с помощью команды:

```
git clone https://github.com/MansurYa/labs-for-algorithms-and-data-st
```

Перейдите в директорию проекта:

```
cd labs-for-algorithms-and-data-structures/Lab4
```

Установите необходимые зависимости:

```
pip install -r requirements.txt
```

6.3 Запуск программы и выполнение задач

Запустите программу с помощью команды:

```
python3 main.py
```

Нажмите кнопку **«Старт»** для начала тестирования.

6.4 Результаты тестирования программы

Тестирование проводилось без ограничения скорости. Результаты представлены в таблице ниже:

Table 2: Результаты тестирования программы

Количество поколений	**Количество вычислений целевой функции**	**Наилучшее решение**	**Значение в лучшей точке**
5	150	(0.15, -0.34)	0.578
10	300	(-0.0129, 0.0564)	0.0143
20	600	(0.0014, -0.0017)	0.000021
40	1200	(0.0000, 0.0000)	0.000000

6.5 Заключение

Результаты тестирования показывают, что алгоритм роя частиц эффективно находит оптимальные решения, демонстрируя высокую точность при увеличении числа итераций.

7 Анализ и улучшение алгоритма

7.1 Улучшение алгоритма

1. **Тонкая настройка параметров**:

   - Регулировка коэффициентов инерции и влияния позволяет достичь баланса между глобальным и локальным поиском, улучшая способность алгоритма находить глобальные оптимумы.

2. **Адаптивное ограничение скорости**:

   - Введение динамического ограничения скорости, изменяющегося в зависимости от состояния роя, способствует более адаптивному поведению алгоритма.

3. **Гибридизация и улучшение инициализации**:

   - Комбинирование PSO с другими методами и использование стратегий, таких как латинский гиперкуб, для начальной инициализации, делают алгоритм более универсальным.

7.2 Результаты тестирования программы

Тестирование проводилось с ограничением скорости = 2.0. Результаты представлены в таблице ниже:

Table 3: Результаты тестирования программы

Количество поколений	**Количество вычислений целевой функции**	**Наилучшее решение**	**Значение в лучшей точке**
5	150	(-0.14, 0.15)	0.18
10	300	(0.0009, 0.0313)	0.005
20	600	(0.0004, 0.0002)	0.000001
40	1200	(0.0000, 0.0000)	0.000000

7.3 Анализ результатов

1. **Без ограничения скорости**:

    - Алгоритм демонстрирует быструю сходимость, но может быть нестабилен на начальных этапах. Это проявляется в высоких значениях функции при малом числе поколений. Например, при 5 поколениях значение функции составляет 0.578, что указывает на разброс частиц.

2. **С ограничением скорости**:

    - Ограничение скорости улучшает стабильность и точность, особенно на начальных этапах. Это позволяет избежать преждевременной сходимости и улучшает исследование пространства решений. Например, при 5 поколениях значение функции снижается до 0.18.

8 Сравнение ГА с алгоритмом роевого интеллекта

8.1 Результаты тестирования программы

Тестирование проводилось с ограничением скорости = 2.0. Результаты представлены в таблице ниже:

Table 4: Результаты тестирования программы

Количество поколений	**Количество вычислений целевой функции**	**Наилучшее решение**	**Значение в лучшей точке**
5	600	(-8.2e-04, -8.5e-04)	6e-06
10	1100	(-2.6e-06, -1.9e-06)	0.000000
20	2100	(-2.2e-10, -9.1e-9)	0.000000
40	4100	(-2.2e-13, -9.1e-10)	0.000000

8.2 Точность

PSO:

- **С ограничением скорости**: Достигает высокой точности, особенно при увеличении числа поколений. Например, при 40 поколениях значение функции достигает 0.000000.
- **Без ограничения скорости**: Может быть менее точным на начальных этапах, но также достигает высокой точности при увеличении числа поколений.

GA:

- Обеспечивает высокую точность благодаря генетическим операциям, особенно при большом числе поколений. Например, уже при 10 поколениях значение функции приближается к нулю.

8.3 Скорость сходимости

PSO:

- Быстро достигает приемлемых решений благодаря своей простой структуре и параллельной природе. Это особенно заметно при малом числе поколений.

GA:

- Медленнее из-за сложных операций, но может быть более эффективным в долгосрочной перспективе.

8.4 Вычислительная сложность

PSO:

- Менее ресурсоёмкий, так как не использует сложные генетические операции. Это позволяет ему быстрее обрабатывать данные.

GA:

- Более ресурсоёмкий из-за необходимости обработки больших популяций и сложных операций.

8.5 Устойчивость к локальным минимумам

PSO:

- Может застревать в локальных минимумах, особенно без ограничения скорости.

GA:

- Благодаря мутации и кроссоверу, лучше избегает локальных минимумов.

8.6 Простота реализации

PSO:

- Проще в реализации и настройке, требует меньше параметров.

GA:

- Сложнее из-за необходимости выбора и настройки различных операторов.

8.7 Заключение

- **PSO** подходит для задач, где важна быстрая сходимость и ограниченные вычислительные ресурсы. Он прост в реализации и эффективен в многомерных пространствах.
- **GA** более универсален и точен, особенно в задачах со сложными ландшафтами, но требует больше времени и ресурсов.

Выбор между PSO и GA зависит от конкретных требований задачи, доступных ресурсов и приоритетов. В некоторых случаях может быть полезно использовать гибридные подходы, комбинируя PSO и GA для достижения лучших результатов.

9 Выводы по работе

В ходе исследования были изучены и сравнены два алгоритма оптимизации: алгоритм роя частиц (PSO) и генетический алгоритм (GA).

- **Алгоритм роя частиц (PSO)** продемонстрировал быструю сходимость и высокую точность при правильной настройке параметров. Он оказался эффективным для задач, требующих быстрого получения результатов и ограниченных вычислительных ресурсов.
- **Генетический алгоритм (GA)** обеспечил устойчивость к локальным минимумам и высокую точность, но потребовал больше времени и ресурсов. Он подходит для сложных задач с высокими требованиями к точности.

Рекомендуется использовать PSO для задач, где важна скорость, а GA — для задач, требующих высокой точности и устойчивости. Эти выводы помогут в выборе подходящего алгоритма в зависимости от специфики задачи и доступных ресурсов.