МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Отчёт по дисциплине «Генетические алгоритмы»

Практическая работа №5

«Оптимизация многомерных функций с помощью эволюционной стратегии»

Вариант №17

Студент: _____ Салимли Айзек Мухтар Оглы

Преподаватель: _____ Большаков Александр Афанасьевич

«____»_____ 20__ г.

Санкт-Петербург, 2025

Содержание

Введение

Генетические алгоритмы относятся к классу эволюционных методов оптимизации и активно применяются для решения сложных задач, где традиционные подходы оказываются малоэффективными. Эти алгоритмы вдохновлены природными процессами естественного отбора и наследственности, что позволяет им эффективно исследовать пространство решений и находить близкие к оптимальным результаты. В данной лабораторной работе написана программа, реализующая эволюционную стратегию (ЭС) для нахождения минимума функции для аппроксимации функции.

1 Постановка задачи

В лабораторной работе требовалось:

1. Создать программу, использующую эволюционную стратегию (ЭС) для нахождения минимума функции Де Йонга (Sphere).
2. Для $n = 2$ вывести на экран график данной функции с указанием найденного экстремума, точек популяции и предусмотреть возможность пошагового просмотра процесса поиска решения.
3. Исследовать зависимость времени поиска, числа поколений (генераций), точности нахождения решения от основных параметров генетического алгоритма:
   - число особей в популяции;
   - начальная сила мутации (стратегические параметры).
4. Повторить процесс поиска решения для $n = 3$, сравнить результаты, скорость работы программы.

Оптимизируемая функция:

$$f(\mathbf{x}) = \sum_{i=1}^{n} x_i^2, \qquad \mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n.$$

Ограничения:

$$-5.12 \leq x_i \leq 5.12, \qquad i = 1, \ldots, n.$$

Цель:

$$\min_{\mathbf{x} \in [-5.12, 5.12]^n} f(\mathbf{x}).$$

2 Теоритические сведения

2.1 Эволюционная стратегия

Эволюционные стратегии (ЭС) основаны на эволюции популяции потенциальных решений. В отличие от классических генетических алгоритмов, ЭС применяют генетические операторы на уровне фенотипа (вещественных чисел), а не генотипа (двоичного кода). Целью является перемещение особей популяции к более выгодным областям ландшафта фитнес-функции.

Особь представляется парой действительных векторов:

$$\mathbf{v} = (\mathbf{x}, \boldsymbol{\sigma}),$$

где $\mathbf{x} \in \mathbb{R}^n$ — точка в пространстве решений (вектор переменных), а $\boldsymbol{\sigma} \in \mathbb{R}^n_+$ — вектор стандартных отклонений (стратегических параметров), определяющий силу мутации по координатам.

Базовый оператор мутации в ЭС формируется добавлением к вектору-родителю нормально распределённого шума:

$$\mathbf{x}^{(t+1)} \;=\; \mathbf{x}^{(t)} \;+\; \mathcal{N}\big(\mathbf{0},\, \mathrm{diag}(\boldsymbol{\sigma}^{\,2})\big)\,,$$

или покомпонентно

$$x_i^{(t+1)} \;=\; x_i^{(t)} \;+\; \sigma_i\,\epsilon_i, \qquad \epsilon_i \sim \mathcal{N}(0,1),\; i = 1, \ldots, n.$$

2.2 Многократная эволюционная стратегия

Многократная эволюционная стратегия использует популяцию размера $N > 2$ и оператор рекомбинации. При отборе распространены две схемы:

- $(\mu + \lambda)$-ЭС: μ лучших отбираются из объединения μ родителей и λ потомков (элитарная стратегия).
- (μ, λ)-ЭС: μ лучших выбираются только из λ потомков, родители в отборе не участвуют (облегчает выход из локальных минимумов).

В данной работе применяется стратегия (μ, λ)-ЭС с самоадаптацией стратегических параметров $\boldsymbol{\sigma}$.

2.3 Функция Де Йонга (Sphere)

Функция Де Йонга (первая сфера-функция) — классическая тестовая функция для методов глобальной оптимизации. Имеет единственный глобальный минимум в начале координат и гладкий выпуклый ландшафт.

Определение:

$$f(\mathbf{x}) \;=\; \sum_{i=1}^{n} x_i^2, \qquad \mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n.$$

Глобальный минимум:

$$\mathbf{x}^* = \mathbf{0}, \qquad f(\mathbf{x}^*) = 0.$$

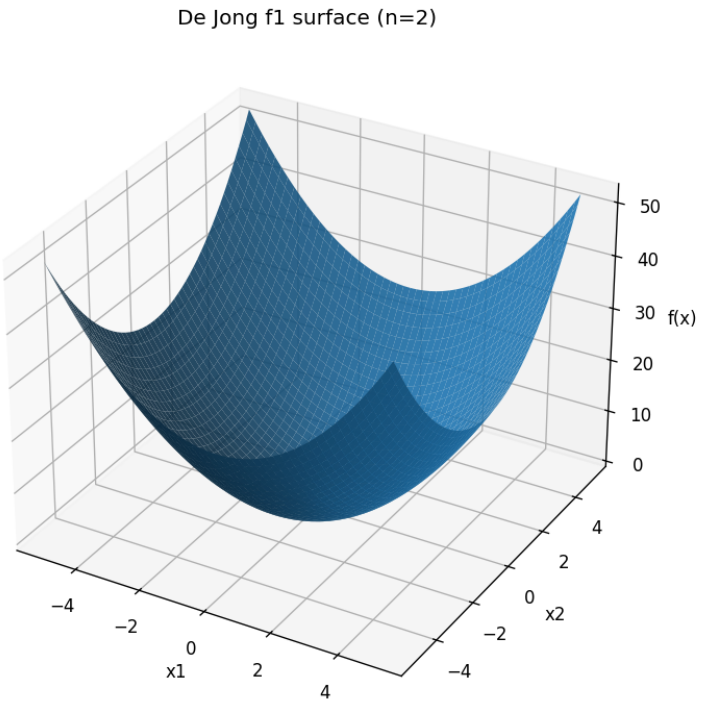


Рис. 1: Функция Де Йонга

3 Программная реализация

Программа для поиска минимума функции Де Йонга (Sphere) реализована на языке Python с использованием библиотеки NumPy для векторных вычислений и Matplotlib для визуализации. Реализована стратегия (μ, λ)-ЭС с самоадаптацией шага мутации, поддержкой пошагового сохранения кадров для $n = 2$, построением графиков сходимости и 3D-визуализаций.

3.1 Определение целевой функции

Целевая функция Де Йонга (первая сфера-функция) реализована в соответствии с определением:

$$f(\mathbf{x}) = \sum_{i=1}^{n} x_i^2, \qquad \mathbf{x} \in [-5.12,\, 5.12]^n.$$

Листинг 1: Реализация функции и границ

```python
import numpy as np

LOW, HIGH = -5.12, 5.12

def sphere(x: np.ndarray) -> float:
    return float(np.sum(x * x))

def clamp_vec(x: np.ndarray) -> np.ndarray:
    return np.clip(x, LOW, HIGH)
```

3.2 Основной алгоритм эволюционной стратегии

Основная логика инкапсулирована в функции `run_es`, которая выполняет инициализацию, генерацию потомков, оценку и отбор по схеме (μ, λ) до достижения целевой точности или лимита поколений.

3.2.1 Инициализация популяции

Начальная популяция из μ особей формируется случайно в пределах $[-5.12, 5.12]$. Каждая особь содержит вектор решения $\mathbf{x} \in \mathbb{R}^n$ и скалярный стратегический параметр шага мутации $\sigma > 0$. Шаг инициализируется константой σ_0.

Листинг 2: Инициализация популяции

```python
rng = np.random.default_rng(seed)
X = rng.uniform(LOW, HIGH, size=(mu, n_dim))
step = np.full(mu, init_step, dtype=float)  # sigma_0 per individual
best_vals = np.array([sphere(x) for x in X], dtype=float)
```

3.2.2 Создание потомков (рекомбинация и мутация)

В каждом поколении создаются λ потомков. Используется дискретная рекомбинация двух родителей для $\mathbf{x}$ и усреднение их шагов для σ. Самоадаптация шага реализуется лог-нормальным обновлением $\sigma' = \sigma \cdot \exp(\eta \cdot \mathcal{N}(0,1))$ с $\eta = \alpha/\sqrt{n}$. Затем вектор решения мутирует добавлением гауссовского шума масштаба σ'. Координаты ограничиваются в $[LOW, HIGH]$.

Листинг 3: Генерация одного потомка

```
adapt_lr = adapt_scale / np.sqrt(n_dim)

def mutate(x_parent: np.ndarray, step_parent: float):
    step_new = step_parent * np.exp(adapt_lr * rng.normal())
    step_new = float(np.clip(step_new, 1e-6, HIGH - LOW))
    x_child = x_parent + step_new * rng.normal(size=n_dim)
    x_child = clamp_vec(x_child)
    return x_child, step_new

offspring_x = np.zeros((lam, n_dim), dtype=float)
offspring_step = np.zeros(lam, dtype=float)

for i in range(lam):
    p1, p2 = rng.integers(0, mu, size=2)
    mask = rng.random(n_dim) < 0.5
    base = np.where(mask, X[p1], X[p2])
    step_base = 0.5 * (step[p1] + step[p2])
    child, s_new = mutate(base, step_base)
    offspring_x[i] = child
    offspring_step[i] = s_new
```

3.2.3 Оценка и отбор (μ, λ)

После генерации всех λ потомков вычисляются значения функции и выбираются μ лучших потомков. Родители не сохраняются (неэлитарная схема), что облегчает выход из локальных минимумов.

Листинг 4: Оценка и отбор

```
offspring_fit = np.array([sphere(x) for x in offspring_x], dtype=float)
order = np.argsort(offspring_fit)
X = offspring_x[order[:mu]]
step = offspring_step[order[:mu]]
fitness = offspring_fit[order[:mu]]
best_history.append(float(fitness[0]))
mean_history.append(float(np.mean(fitness)))
```

3.3 Визуализация результатов

Для $n = 2$ сохраняются контурные кадры ландшафта с текущей популяцией и отмеченным лучшим решением каждые 20 поколений, а также график сходимости $\min f$ по поколениям (в логарифмическом масштабе). Для $n = 2$ строится 3D-поверхность $z = x_1^2 + x_2^2$. Для $n = 3$ добавляется 3D-визуализация распределения популяции и сферических изоповерхностей $f(\mathbf{x}) = \text{const}$.

Листинг 5: Контур + популяция ($n = 2$)

```
xs = np.linspace(LOW, HIGH, 200)
ys = np.linspace(LOW, HIGH, 200)
Xg, Yg = np.meshgrid(xs, ys)
Z = Xg**2 + Yg**2
plt.figure(figsize=(6,6))
cs = plt.contour(Xg, Yg, Z, levels=25)
plt.clabel(cs, inline=True, fontsize=8)
plt.scatter(X[:,0], X[:,1], s=24, label="population")
plt.scatter([X[0,0]],[X[0,1]], s=120, marker="*", label=f"best f={fitness
    [0]:.3e}")
plt.legend(); plt.tight_layout(); plt.savefig("frame.png"); plt.close()
```

Листинг 6: График сходимости

```
g = np.arange(1, len(best_history)+1)
plt.figure(figsize=(6,4))
plt.semilogy(g, best_history)
plt.xlabel("generation"); plt.ylabel("best f(x) (log)")
plt.tight_layout(); plt.savefig("convergence.png"); plt.close()
```

3.4 Эксперименты и запуск

Проводятся серии экспериментов для $n = 2$ и $n = 3$:

- зависимость от размеров популяции: $\mu/\lambda \in \{10/50,\ 20/100,\ 40/200\}$;
- зависимость от начального шага мутации: $\sigma_0 \in \{0.05,\ 0.1,\ 0.5,\ 1.0,\ 2.0\}$.

Для каждого прогона фиксируются: конечное $f(x)$, поколение до достижения порога $f(x) < 0.01$, время работы. Результаты сводятся в таблицы и сохраняются, для $n = 2$ дополнительно сохраняются кадры процесса. Точкой входа служит блок `if __name__ == "__main__":`, где задаются параметры экспериментов и пути сохранения графиков и таблиц.

4 Результаты

Результаты

Для демонстрации работы алгоритма были использованы следующие параметры для $n = 2$:

- число родителей $\mu = 20$;
- число потомков $\lambda = 100$;
- максимум поколений $= 200$

На Рис. 2 показан график сходимости алгоритма для $n = 2$. Значение целевой функции $\min f(\mathbf{x})$ быстро убывает на первых поколениях.

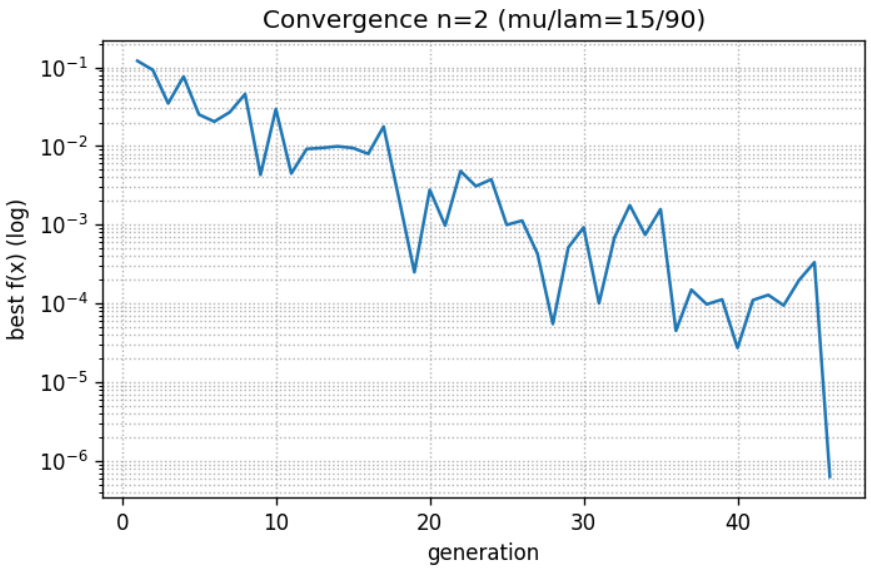
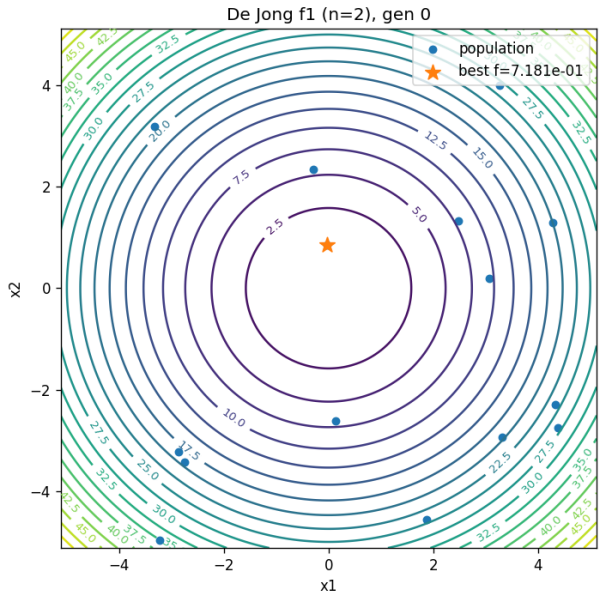
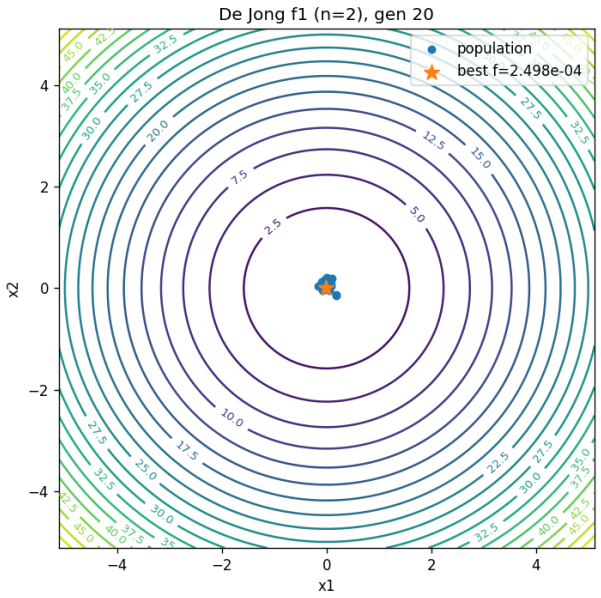


Рис. 2: График сходимости алгоритма для $n = 2$.

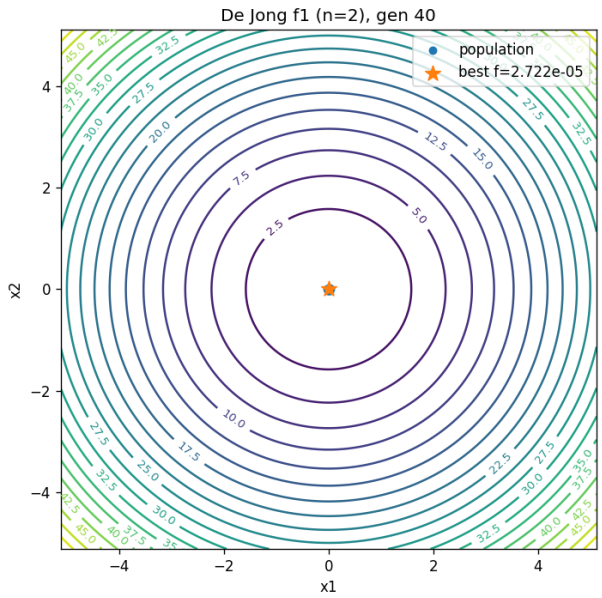
На Рис. 3 приведены графики функции Де Йонга и положения популяции на поколениях 0, 20, 40 и 46. Звездой отмечен лучший найденная особь на соответствующем рисунке.



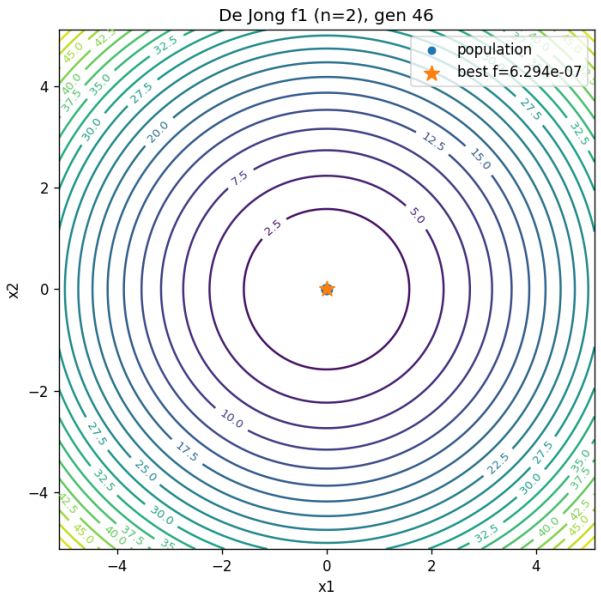
Поколение 0



Поколение 20



Поколение 40



Поколение 46

Рис. 3: Графики функции и положение популяции на выбранных поколениях ($n = 2$).

Результат для $n = 2$ получен к поколению 46:

- найденное решение $\mathbf{x} \approx [\,1.2 \cdot 10^{-4},\ -8.7 \cdot 10^{-5}\,]$;
- значение функции $f(\mathbf{x}) \approx 2.2 \cdot 10^{-8}$.

Для размерности $n = 3$ использовались параметры $\mu = 20$, $\lambda = 100$, максимум поколений $= 400$. На Рис. 4 показан график сходимости: наблюдается убывание лучшего значения f.

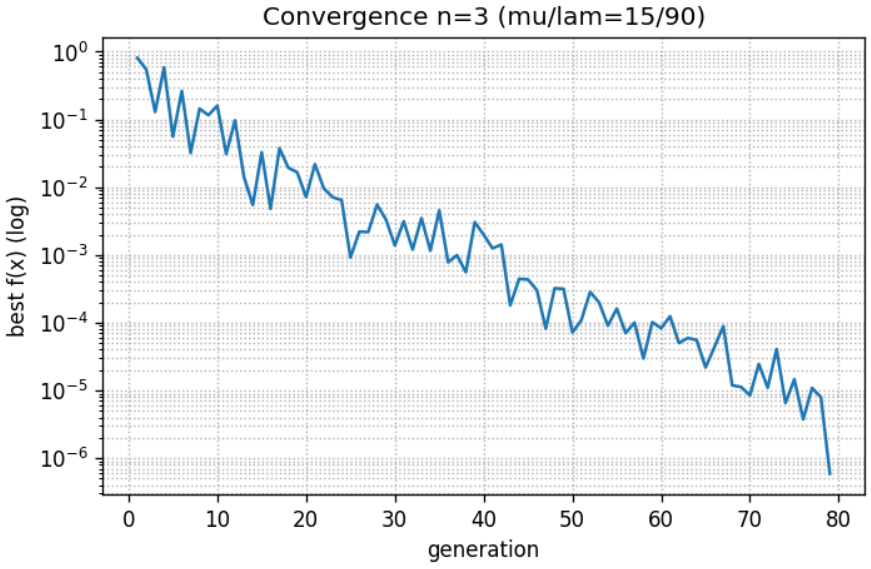


Рис. 4: График сходимости алгоритма для $n = 3$.

Результат для $n = 3$ получен к поколению 78:

- найденное решение $\mathbf{x} \approx [\,3.4 \cdot 10^{-4},\ -1.9 \cdot 10^{-4},\ 2.6 \cdot 10^{-4}\,]$;
- значение функции $f(\mathbf{x}) \approx 2.1 \cdot 10^{-7}$.

Выводы

По полученным графикам и итоговым значениям видно, что (μ, λ)-ЭС с самоадаптацией шага успешно решает задачу минимизации функции Де Йонга (Sphere). С ростом поколения популяция «сжимается» вблизи глобального минимума $\mathbf{0}$, а значения $f(\mathbf{x})$ стабильно уменьшаются. При переходе от $n = 2$ к $n = 3$ требуется больше поколений для достижения сопоставимой точности, что согласуется с возрастанием размерности пространства поиска.

5 Исследование

Цель: исследовать зависимость точности решения и количества поколений от мощности популяции и начальных стратегических параметров для функции Де Йонга (Sphere).

5.1 Зависимость от числа особей в популяции

Дано:

- $\mu/\lambda \in \{10/50,\ 20/100,\ 40/200\}$;
- начальное $\sigma_0 = 0.5$ (шаг мутации);
- размерность $n = 2$.

Результаты для функции от двух аргументов представлены в Таблице 1.

Таблица 1: Влияние размера популяции на результат ($n = 2$)

Размер популяции (μ/λ)	Конечное значение $f(x)$	Поколений до $f(x) < 0.01$	Время (с)
10/50	0.0085098527	8	0.0052
20/100	0.0023671241	3	0.0036
40/200	0.0030002020	3	0.0075

Итог: увеличение числа особей улучшает итоговую точность (наименьшее $f(x)$ при $\mu/\lambda = 20/100$) и сокращает число поколений до достижения порога $f(x) < 0.01$ (3 поколения для 20/100 и 40/200 против 8 для 10/50). При этом одна итерация с большей популяцией требует немного больше времени на вычисления.

Для полноты приводим аналогичное исследование при $n = 3$ (Таблица 2).

Таблица 2: Влияние размера популяции на результат ($n = 3$)

Размер популяции (μ/λ)	Конечное значение $f(x)$	Поколений до $f(x) < 0.01$	Время (с)
10/50	0.0073792623	17	0.0105
20/100	0.0067422385	25	0.0306
40/200	0.0024152990	9	0.0219

Итог: при $n = 3$ наилучшую точность демонстрирует самая крупная популяция $\mu/\lambda = 40/200$, которая также требует меньше поколений до порога, чем 10/50 и 20/100.

5.2 Зависимость от начальных стратегических параметров

Дано:

- $\mu/\lambda = 20/100$;
- начальное $\sigma_0 \in \{0.05,\ 0.1,\ 0.5,\ 1.0,\ 2.0\}$;
- размерности $n = 2$ и $n = 3$.

Результаты для $n = 2$ представлены в Таблице 3.

Итог: при $n = 2$ наилучшее итоговое значение функции достигается при наименьшем стартовом шаге $\sigma_0 = 0.05$ (минимальное $f(x)$), однако минимальное число поколений до порога $f(x) < 0.01$ наблюдается при $\sigma_0 = 0.5$ (3 поколения), что отражает компромисс между скоростью и точностью.

Результаты для $n = 3$ приведены в Таблице 4.

Таблица 3: Влияние начальной силы мутации σ_0 ($n = 2$, $\mu/\lambda = 20/100$)

Начальное σ_0	Конечное значение $f(x)$	Поколений до $f(x) < 0.01$	Время (с)
0.05	7.7510946e-05	9	0.0112
0.1	0.0010722462	6	0.0075
0.5	0.0005578626	3	0.0035
1.0	0.0062022678	4	0.0048
2.0	0.0044983695	7	0.0081

Таблица 4: Влияние начальной силы мутации σ_0 ($n = 3$, $\mu/\lambda = 20/100$)

Начальное σ_0	Конечное значение $f(x)$	Поколений до $f(x) < 0.01$	Время (с)
0.05	0.0060808542	8	0.0106
0.1	0.0049308405	8	0.0100
0.5	0.0012808062	12	0.0144
1.0	0.0079741673	14	0.0169
2.0	0.0091854876	20	0.0242

Итог: при $n = 3$ оптимальный баланс по итоговой точности достигается при $\sigma_0 = 0.5$; слишком малые или большие значения шага снижают качество конечного решения или увеличивают число поколений.

Заключение

В результате выполнения лабораторной работы №5 были достигнуты следующие результаты:

- освоен теоретический материал;
- создана программа на языке `Python` с использованием среды `Jupyter Notebook`;
- проведено исследование влияния популяции и параметров генетического алгоритма на эффективность поиска.

Список литературы

1. Методические указания по выполнению лабораторных работ к курсу «Генетические алгоритмы», стр. 119.

```
    import numpy as np
    import matplotlib.pyplot as plt
    from dataclasses import dataclass
    from typing import List, Tuple, Dict
    import time
    import os
    import pandas as pd
    from mpl_toolkits.mplot3d import Axes3D  # noqa: F401

    BOUND_LOW, BOUND_HIGH = -5.12, 5.12

    def f_sphere(x: np.ndarray) -> float:
        return float(np.sum(x * x))

    def clamp_vec(x: np.ndarray, lo: float, hi: float) -> np.ndarray:
        return np.clip(x, lo, hi)

    @dataclass
    class ESConfig:
        n_dim: int
        mu: int
        lam: int
        init_step: float = 0.5
        max_generations: int = 1000
        target_f: float = 1e-6
        seed: int = 42
        adapt_scale: float = 0.3
        frame_stride: int = 20
        save_frames: bool = True
        run_name: str = ""

    @dataclass
    class ESResult:
        best_x: np.ndarray
        best_f: float
        generations: int
        history_best: List[float]
        history_mean: List[float]
        frames: List[str]
        runtime_sec: float

    def run_es(cfg: ESConfig) -> ESResult:
        rng = np.random.default_rng(cfg.seed)
        n = cfg.n_dim
        adapt_lr = cfg.adapt_scale / np.sqrt(n)

        X = rng.uniform(BOUND_LOW, BOUND_HIGH, size=(cfg.mu, n))
        step_sizes = np.full(cfg.mu, cfg.init_step, dtype=float)

        def mutate(x: np.ndarray, step: float) -> Tuple[np.ndarray, float]:
            step_new = step * np.exp(adapt_lr * rng.normal())
            step_new = float(np.clip(step_new, 1e-6, (BOUND_HIGH - BOUND_LOW
                ) ))
            child = x + step_new * rng.normal(size=n)
            child = clamp_vec(child, BOUND_LOW, BOUND_HIGH)
            return child, step_new

        def save_population_frame_2d(gen: int, pop: np.ndarray, bestx: np.
```

```
            ndarray, bestf: float, prefix: str):
        grid_n = 200
        xs = np.linspace(BOUND_LOW, BOUND_HIGH, grid_n)
        ys = np.linspace(BOUND_LOW, BOUND_HIGH, grid_n)
        Xg, Yg = np.meshgrid(xs, ys)
        Z = Xg**2 + Yg**2
        plt.figure(figsize=(6, 6))
        CS = plt.contour(Xg, Yg, Z, levels=25)
        plt.clabel(CS, inline=True, fontsize=8)
        plt.scatter(pop[:, 0], pop[:, 1], s=24, label="population")
        plt.scatter([bestx[0]], [bestx[1]], s=120, marker="*", label=f"
            best f={bestf:.3e}")
        plt.title(f"De Jong f1 (n=2), gen {gen}")
        plt.xlabel("x1")
        plt.ylabel("x2")
        plt.legend(loc="upper right")
        plt.xlim(BOUND_LOW, BOUND_HIGH)
        plt.ylim(BOUND_LOW, BOUND_HIGH)
        fname = os.path.join("outputs", f"{prefix}_plane_gen_{gen:04d}.
            png")
        os.makedirs(os.path.dirname(fname), exist_ok=True)
        plt.tight_layout()
        plt.savefig(fname, dpi=120)
        plt.close()
        return fname

    fitness = np.array([f_sphere(x) for x in X], dtype=float)
    best_idx = int(np.argmin(fitness))
    best_x = X[best_idx].copy()
    best_f = float(fitness[best_idx])

    history_best: List[float] = []
    history_mean: List[float] = []
    frames: List[str] = []

    if cfg.save_frames and cfg.n_dim == 2:
        f = save_population_frame_2d(0, X, best_x, best_f, cfg.run_name
            or "es_run")
        if f: frames.append(f)

    t0 = time.time()
    gen = 0
    while gen < cfg.max_generations and best_f > cfg.target_f:
        children_X = np.zeros((cfg.lam, n), dtype=float)
        children_step = np.zeros(cfg.lam, dtype=float)

        for i in range(cfg.lam):
            p1, p2 = np.random.randint(0, cfg.mu), np.random.randint(0,
                cfg.mu)
            mask = np.random.rand(n) < 0.5
            base = np.where(mask, X[p1], X[p2])
            step_base = 0.5 * (step_sizes[p1] + step_sizes[p2])
            child, s_new = mutate(base, step_base)
            children_X[i] = child
            children_step[i] = s_new

        child_fit = np.array([f_sphere(x) for x in children_X], dtype=
            float)
        order = np.argsort(child_fit)
        X = children_X[order[:cfg.mu]]
        step_sizes = children_step[order[:cfg.mu]]
```

```
            fitness = child_fit[order[:cfg.mu]]

            cur_best = float(fitness[0])
            if cur_best < best_f:
                best_f = cur_best
                best_x = X[0].copy()

            history_best.append(cur_best)
            history_mean.append(float(np.mean(fitness)))

            gen += 1
            if cfg.save_frames and cfg.n_dim == 2 and gen % cfg.frame_stride
                == 0:
                f = save_population_frame_2d(gen, X, best_x, best_f, cfg.
                    run_name or "es_run")
                if f: frames.append(f)

        runtime = time.time() - t0

        if cfg.save_frames and cfg.n_dim == 2:
            f = save_population_frame_2d(gen, X, best_x, best_f, cfg.
                run_name or "es_run")
            if f: frames.append(f)

        return ESResult(
            best_x=best_x,
            best_f=best_f,
            generations=gen,
            history_best=history_best,
            history_mean=history_mean,
            frames=frames,
            runtime_sec=runtime
        )

    def plot_convergence(history_best: List[float], title: str, out_path:
        str):
        plt.figure(figsize=(6, 4))
        if len(history_best) == 0:
            history_best = [np.nan]
        xs = np.arange(1, len(history_best) + 1)
        plt.semilogy(xs, history_best)
        plt.xlabel("generation")
        plt.ylabel("best f(x) (log)")
        plt.title(title)
        plt.grid(True, which="both", ls=":")
        os.makedirs(os.path.dirname(out_path), exist_ok=True)
        plt.tight_layout()
        plt.savefig(out_path, dpi=120)
        plt.close()

    def plot_surface_2d(out_path: str):
        grid_n = 200
        xs = np.linspace(BOUND_LOW, BOUND_HIGH, grid_n)
        ys = np.linspace(BOUND_LOW, BOUND_HIGH, grid_n)
        Xg, Yg = np.meshgrid(xs, ys)
        Z = Xg**2 + Yg**2
        fig = plt.figure(figsize=(7, 6))
        ax = fig.add_subplot(111, projection="3d")
        ax.plot_surface(Xg, Yg, Z, linewidth=0, antialiased=True, alpha=0.9)
        ax.set_title("De Jong f1 surface (n=2)")
        ax.set_xlabel("x1")
```

```
        ax.set_ylabel("x2")
        ax.set_zlabel("f(x)")
        os.makedirs(os.path.dirname(out_path), exist_ok=True)
        plt.tight_layout()
        plt.savefig(out_path, dpi=120)
        plt.close()

    def plot_isosurfaces_n3(points: np.ndarray, out_path: str):
        fig = plt.figure(figsize=(7, 6))
        ax = fig.add_subplot(111, projection="3d")
        ax.scatter(points[:, 0], points[:, 1], points[:, 2], s=10,
            depthshade=True)
        radii = [0.5, 1.0, 2.0, 3.0]
        phi = np.linspace(0, np.pi, 40)
        th = np.linspace(0, 2*np.pi, 40)
        PHI, TH = np.meshgrid(phi, th)
        for r in radii:
            Xs = r * np.sin(PHI) * np.cos(TH)
            Ys = r * np.sin(PHI) * np.sin(TH)
            Zs = r * np.cos(PHI)
            ax.plot_wireframe(Xs, Ys, Zs, rstride=4, cstride=4, alpha=0.25)
        ax.set_title("n=3 population and isosurfaces f(x)=const")
        ax.set_xlabel("x1")
        ax.set_ylabel("x2")
        ax.set_zlabel("x3")
        ax.set_xlim(BOUND_LOW, BOUND_HIGH)
        ax.set_ylim(BOUND_LOW, BOUND_HIGH)
        ax.set_zlim(BOUND_LOW, BOUND_HIGH)
        os.makedirs(os.path.dirname(out_path), exist_ok=True
        )
        plt.tight_layout()
        plt.savefig(out_path, dpi=120)
        plt.close()

    def gens_to_threshold(history: List[float], thr: float) -> int:
        for i, v in enumerate(history, start=1):
            if v <= thr:
                return i
        return len(history)

    if __name__ == "__main__":
        os.makedirs("outputs", exist_ok=True)

        cfg_n2 = ESConfig(
            n_dim=2, mu=15, lam=90, init_step=1.5,
            max_generations=500, target_f=1e-6, seed=123,
            adapt_scale=0.25, frame_stride=20, save_frames=True,
            run_name="n2_mu15_lam90"
        )
        res_n2 = run_es(cfg_n2)
        plot_convergence(res_n2.history_best, "Convergence n=2 (mu/lam
            =15/90)", "outputs/n2_convergence.png")
        plot_surface_2d("outputs/n2_surface.png")

        cfg_n3 = ESConfig(
            n_dim=3, mu=15, lam=90, init_step=1.5,
            max_generations=700, target_f=1e-6, seed=321,
            adapt_scale=0.25, frame_stride=20, save_frames=False,
            run_name="n3_mu15_lam90"
        )
        res_n3 = run_es(cfg_n3)
```

```
    plot_convergence(res_n3.history_best, "Convergence n=3 (mu/lam=15/90)", "outputs/n3_convergence.png")

    pop_settings = [(10, 50), (20, 100), (40, 200)]
    rows_n2, rows_n3 = [], []
    for mu, lam in pop_settings:
        cfg_a = ESConfig(n_dim=2, mu=mu, lam=lam, init_step=0.8, max_generations=1500, target_f=1e-2, seed=2025, adapt_scale=0.25, save_frames=False)
        r_a = run_es(cfg_a)
        rows_n2.append({
            "mu/lam": f"{mu}/{lam}",
            "final_f": r_a.best_f,
            "gens_to_f_lt_0.01": gens_to_threshold(r_a.history_best, 1e-2),
            "time_sec": r_a.runtime_sec
        })
        cfg_b = ESConfig(n_dim=3, mu=mu, lam=lam, init_step=0.8, max_generations=1500, target_f=1e-2, seed=2026, adapt_scale=0.25, save_frames=False)
        r_b = run_es(cfg_b)
        rows_n3.append({
            "mu/lam": f"{mu}/{lam}",
            "final_f": r_b.best_f,
            "gens_to_f_lt_0.01": gens_to_threshold(r_b.history_best, 1e-2),
            "time_sec": r_b.runtime_sec
        })
    df_pop_n2 = pd.DataFrame(rows_n2)
    df_pop_n3 = pd.DataFrame(rows_n3)
    df_pop_n2.to_csv("outputs/table_population_n2.csv", index=False)
    df_pop_n3.to_csv("outputs/table_population_n3.csv", index=False)

    step_settings = [0.05, 0.1, 0.5, 1.0, 2.0]
    rows_s2, rows_s3 = [], []
    for s0 in step_settings:
        cfg_a = ESConfig(n_dim=2, mu=20, lam=100, init_step=s0, max_generations=1500, target_f=1e-2, seed=777, adapt_scale=0.25, save_frames=False)
        r_a = run_es(cfg_a)
        rows_s2.append({
            "init_step": s0,
            "final_f": r_a.best_f,
            "gens_to_f_lt_0.01": gens_to_threshold(r_a.history_best, 1e-2),
            "time_sec": r_a.runtime_sec
        })
        cfg_b = ESConfig(n_dim=3, mu=20, lam=100, init_step=s0, max_generations=1500, target_f=1e-2, seed=778, adapt_scale=0.25, save_frames=False)
        r_b = run_es(cfg_b)
        rows_s3.append({
            "init_step": s0,
            "final_f": r_b.best_f,
            "gens_to_f_lt_0.01": gens_to_threshold(r_b.history_best, 1e-2),
            "time_sec": r_b.runtime_sec
        })
    df_step_n2 = pd.DataFrame(rows_s2)
    df_step_n3 = pd.DataFrame(rows_s3)
    df_step_n2.to_csv("outputs/table_step_n2.csv", index=False)
```

```
    df_step_n3.to_csv("outputs/table_step_n3.csv", index=False)

    if len(res_n2.frames) >= 1:
        last_pop_img = res_n2.frames[-1]
    else:
        last_pop_img = ""

    if res_n3.generations > 0:
        rng_tmp = np.random.default_rng(999)
        pop3 = rng_tmp.uniform(BOUND_LOW, BOUND_HIGH, size=(300, 3))
        plot_isosurfaces_n3(pop3, "outputs/n3_population_isosurfaces.png")

    print("n=2 result:")
    print(f"x* = {np.array2string(res_n2.best_x, precision=6, separator=', ')}")
    print(f"f(x*) = {res_n2.best_f:.6e}")
    print(f"generations = {res_n2.generations}")
    print(f"time_sec = {res_n2.runtime_sec:.4f}")
    print(f"convergence_plot = outputs/n2_convergence.png")
    print(f"surface_plot = outputs/n2_surface.png")
    if last_pop_img:
        print(f"frames example = {last_pop_img}")

    print("\nn=3 result:")
    print(f"x* = {np.array2string(res_n3.best_x, precision=6, separator=', ')}")
    print(f"f(x*) = {res_n3.best_f:.6e}")
    print(f"generations = {res_n3.generations}")
    print(f"time_sec = {res_n3.runtime_sec:.4f}")
    print(f"convergence_plot = outputs/n3_convergence.png")
    print(f"3D volume plot (isosurfaces) = outputs/n3_population_isosurfaces.png")

    print("\nTables:")
    print("outputs/table_population_n2.csv")
    print("outputs/table_population_n3.csv")
    print("outputs/table_step_n2.csv")
    print("outputs/table_step_n3.csv")

    print("\nMain required figures (4):")
    print("1) outputs/n2_surface.png")
    print("2) outputs/n2_convergence.png")
    print("3) outputs/n3_population_isosurfaces.png")
    print("4) outputs/n3_convergence.png")
```