

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.

Содержание

Введение	3
1 Постановка задачи	4
2 Теоритические сведения	5
3 Программная реализация	7
3.1 Определение целевой функции	7
3.2 Основной алгоритм эволюционной стратегии	7
3.2.1 Инициализация популяции	7
3.2.2 Создание потомков (рекомбинация и мутация)	7
3.2.3 Оценка и отбор (μ, λ)	8
3.3 Визуализация результатов	8
3.4 Эксперименты и запуск	9
4 Результаты	10
5 Исследование	13
6 Ответ на контрольный вопрос	15
Заключение	16
Список литературы	17
Приложение А	18

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:

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

Ограничения:

$$-5.12 \leq x_i \leq 5.12, \quad i = 1, \dots, n.$$

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:

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

или покомпонентно

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

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:

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

Глобальный минимум:

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

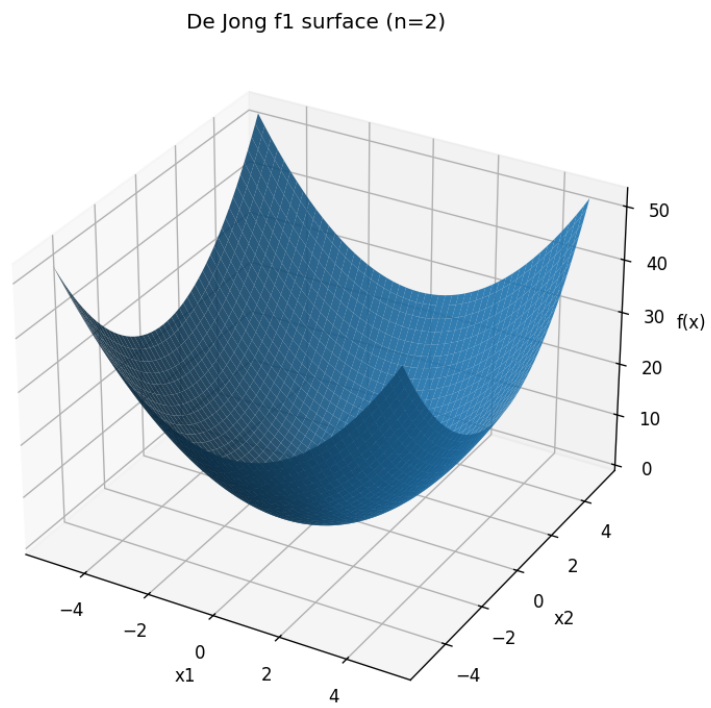


Рис. 1: Функция Де Йонга

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:

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

Листинг 1: Реализация функции и границ

```
1 import numpy as np
2
3 LOW, HIGH = -5.12, 5.12
4
5 def sphere(x: np.ndarray) -> float:
6     return float(np.sum(x * x))
7
8 def clamp_vec(x: np.ndarray) -> np.ndarray:
9     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

```
1 rng = np.random.default_rng(seed)
2 X = rng.uniform(LOW, HIGH, size=(mu, n_dim))
3 step = np.full(mu, init_step, dtype=float) # sigma_0 per individual
4 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

```

1 adapt_lr = adapt_scale / np.sqrt(n_dim)
2
3 def mutate(x_parent: np.ndarray, step_parent: float):
4     step_new = step_parent * np.exp(adapt_lr * rng.normal())
5     step_new = float(np.clip(step_new, 1e-6, HIGH - LOW))
6     x_child = x_parent + step_new * rng.normal(size=n_dim)
7     x_child = clamp_vec(x_child)
8     return x_child, step_new
9
10 offspring_x = np.zeros((lam, n_dim), dtype=float)
11 offspring_step = np.zeros(lam, dtype=float)
12
13 for i in range(lam):
14     p1, p2 = rng.integers(0, mu, size=2)
15     mask = rng.random(n_dim) < 0.5
16     base = np.where(mask, X[p1], X[p2])
17     step_base = 0.5 * (step[p1] + step[p2])
18     child, s_new = mutate(base, step_base)
19     offspring_x[i] = child
20     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

```

1 offspring_fit = np.array([sphere(x) for x in offspring_x], dtype=float)
2 order = np.argsort(offspring_fit)
3 X = offspring_x[order[:mu]]
4 step = offspring_step[order[:mu]]
5 fitness = offspring_fit[order[:mu]]
6 best_history.append(float(fitness[0]))
7 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 ($n = 2$)

```

1 xs = np.linspace(LOW, HIGH, 200)
2 ys = np.linspace(LOW, HIGH, 200)
3 Xg, Yg = np.meshgrid(xs, ys)
4 Z = Xg**2 + Yg**2
5 plt.figure(figsize=(6,6))
6 cs = plt.contour(Xg, Yg, Z, levels=25)
7 plt.clabel(cs, inline=True, fontsize=8)
8 plt.scatter(X[:,0], X[:,1], s=24, label="population")
9 plt.scatter([X[0,0]], [X[0,1]], s=120, marker="*", label=f"best f={fitness
10               [0]:.3e}")
11 plt.legend(); plt.tight_layout(); plt.savefig("frame.png"); plt.close()

```


Листинг 6: График сходимости

```
1 g = np.arange(1, len(best_history)+1)
2 plt.figure(figsize=(6,4))
3 plt.semilogy(g, best_history)
4 plt.xlabel("generation"); plt.ylabel("best f(x) (log)")
5 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.

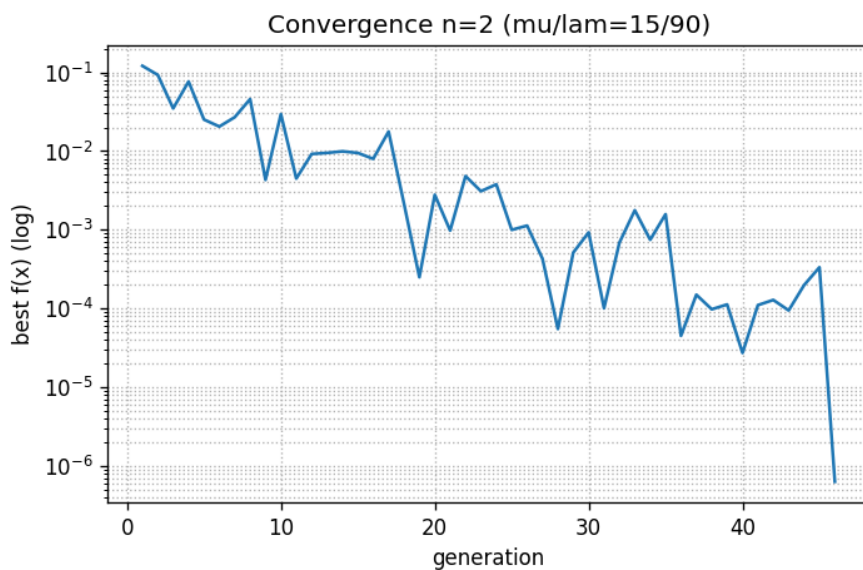


Рис. 2: График сходимости алгоритма для $n = 2$.

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.

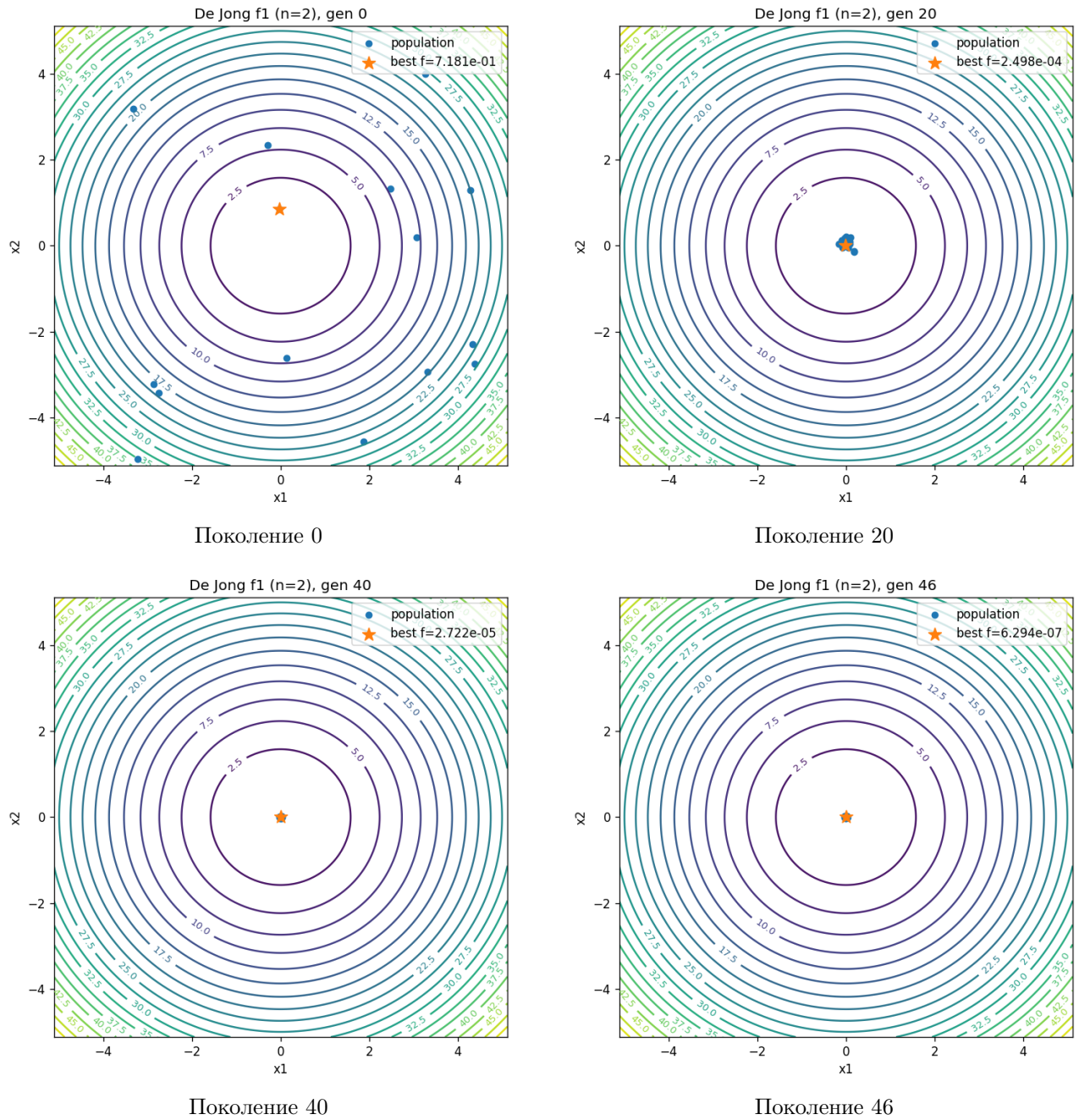


Рис. 3: Графики функции и положение популяции на выбранных поколениях ($n = 2$).

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 .

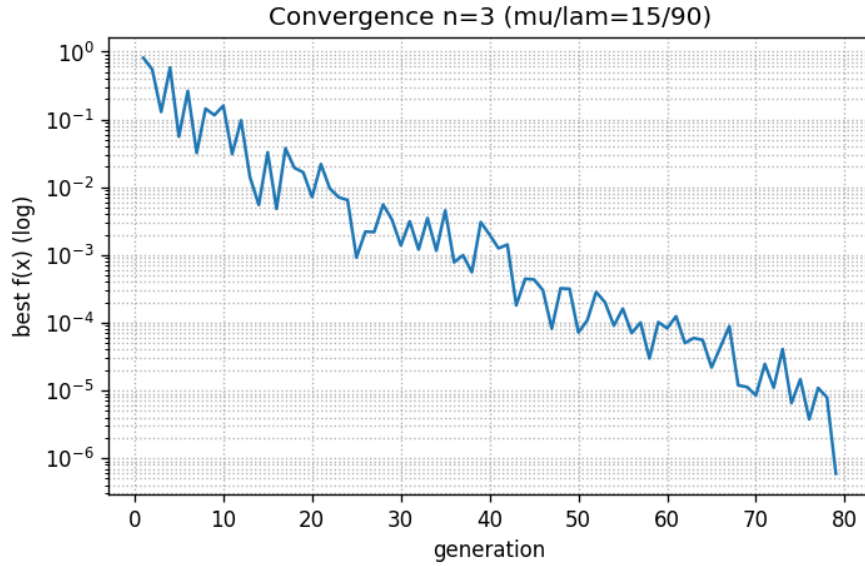


Рис. 4: График сходимости алгоритма для $n = 3$.

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.

6 Ответ на контрольный вопрос

Вопрос: Как выполняется оператор мутации в ЭС?

Ответ:

Основной принцип мутации Особь представляется парой векторов:

$$v = (x, \sigma)$$

где:

- x – точка в пространстве решений (фенотип)
- σ – вектор стандартных отклонений (стратегические параметры)

Формула мутации

$$x^{t+1} = x^t + N(0, \sigma)$$

где:

- $N(0, \sigma)$ – вектор независимых случайных чисел, распределенных по нормальному закону Гаусса с нулевым средним и стандартным отклонением σ
- x^t – родительская особь в поколении t
- x^{t+1} – потомок после мутации

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.

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3  from dataclasses import dataclass
4  from typing import List, Tuple, Dict
5  import time
6  import os
7  import pandas as pd
8  from mpl_toolkits.mplot3d import Axes3D # noqa: F401
9
10 BOUND_LOW, BOUND_HIGH = -5.12, 5.12
11
12 def f_sphere(x: np.ndarray) -> float:
13     return float(np.sum(x * x))
14
15 def clamp_vec(x: np.ndarray, lo: float, hi: float) -> np.ndarray:
16     return np.clip(x, lo, hi)
17
18 @dataclass
19 class ESConfig:
20     n_dim: int
21     mu: int
22     lam: int
23     init_step: float = 0.5
24     max_generations: int = 1000
25     target_f: float = 1e-6
26     seed: int = 42
27     adapt_scale: float = 0.3
28     frame_stride: int = 20
29     save_frames: bool = True
30     run_name: str = ""
31
32 @dataclass
33 class ESResult:
34     best_x: np.ndarray
35     best_f: float
36     generations: int
37     history_best: List[float]
38     history_mean: List[float]
39     frames: List[str]
40     runtime_sec: float
41
42 def run_es(cfg: ESConfig) -> ESResult:
43     rng = np.random.default_rng(cfg.seed)
44     n = cfg.n_dim
45     adapt_lr = cfg.adapt_scale / np.sqrt(n)
46
47     X = rng.uniform(BOUND_LOW, BOUND_HIGH, size=(cfg.mu, n))
48     step_sizes = np.full(cfg.mu, cfg.init_step, dtype=float)
49
50     def mutate(x: np.ndarray, step: float) -> Tuple[np.ndarray, float]:
51         step_new = step * np.exp(adapt_lr * rng.normal())
52         step_new = float(np.clip(step_new, 1e-6, (BOUND_HIGH - BOUND_LOW)))
53         child = x + step_new * rng.normal(size=n)
54         child = clamp_vec(child, BOUND_LOW, BOUND_HIGH)
55         return child, step_new
56
57     def save_population_frame_2d(gen: int, pop: np.ndarray, bestx: np.

```

```

ndarray, bestf: float, prefix: str):
58     grid_n = 200
59     xs = np.linspace(BOUND_LOW, BOUND_HIGH, grid_n)
60     ys = np.linspace(BOUND_LOW, BOUND_HIGH, grid_n)
61     Xg, Yg = np.meshgrid(xs, ys)
62     Z = Xg**2 + Yg**2
63     plt.figure(figsize=(6, 6))
64     CS = plt.contour(Xg, Yg, Z, levels=25)
65     plt.clabel(CS, inline=True, fontsize=8)
66     plt.scatter(pop[:, 0], pop[:, 1], s=24, label="population")
67     plt.scatter([bestx[0]], [bestx[1]], s=120, marker="*", label=f"
        best f={bestf:.3e}")
68     plt.title(f"De Jong f1 (n=2), gen {gen}")
69     plt.xlabel("x1")
70     plt.ylabel("x2")
71     plt.legend(loc="upper right")
72     plt.xlim(BOUND_LOW, BOUND_HIGH)
73     plt.ylim(BOUND_LOW, BOUND_HIGH)
74     fname = os.path.join("outputs", f"{prefix}_plane_gen_{gen:04d}.
        png")
75     os.makedirs(os.path.dirname(fname), exist_ok=True)
76     plt.tight_layout()
77     plt.savefig(fname, dpi=120)
78     plt.close()
79     return fname
80
81     fitness = np.array([f_sphere(x) for x in X], dtype=float)
82     best_idx = int(np.argmin(fitness))
83     best_x = X[best_idx].copy()
84     best_f = float(fitness[best_idx])
85
86     history_best: List[float] = []
87     history_mean: List[float] = []
88     frames: List[str] = []
89
90     if cfg.save_frames and cfg.n_dim == 2:
91         f = save_population_frame_2d(0, X, best_x, best_f, cfg.run_name
            or "es_run")
92         if f: frames.append(f)
93
94     t0 = time.time()
95     gen = 0
96     while gen < cfg.max_generations and best_f > cfg.target_f:
97         children_X = np.zeros((cfg.lam, n), dtype=float)
98         children_step = np.zeros(cfg.lam, dtype=float)
99
100         for i in range(cfg.lam):
101             p1, p2 = np.random.randint(0, cfg.mu), np.random.randint(0,
                cfg.mu)
102             mask = np.random.rand(n) < 0.5
103             base = np.where(mask, X[p1], X[p2])
104             step_base = 0.5 * (step_sizes[p1] + step_sizes[p2])
105             child, s_new = mutate(base, step_base)
106             children_X[i] = child
107             children_step[i] = s_new
108
109         child_fit = np.array([f_sphere(x) for x in children_X], dtype=
            float)
110         order = np.argsort(child_fit)
111         X = children_X[order[:cfg.mu]]
112         step_sizes = children_step[order[:cfg.mu]]

```

```

113         fitness = child_fit[order[:cfg.mu]]
114
115         cur_best = float(fitness[0])
116         if cur_best < best_f:
117             best_f = cur_best
118             best_x = X[0].copy()
119
120         history_best.append(cur_best)
121         history_mean.append(float(np.mean(fitness)))
122
123         gen += 1
124         if cfg.save_frames and cfg.n_dim == 2 and gen % cfg.frame_stride
           == 0:
125             f = save_population_frame_2d(gen, X, best_x, best_f, cfg.
               run_name or "es_run")
126             if f: frames.append(f)
127
128         runtime = time.time() - t0
129
130         if cfg.save_frames and cfg.n_dim == 2:
131             f = save_population_frame_2d(gen, X, best_x, best_f, cfg.
               run_name or "es_run")
132             if f: frames.append(f)
133
134         return ESResult(
135             best_x=best_x,
136             best_f=best_f,
137             generations=gen,
138             history_best=history_best,
139             history_mean=history_mean,
140             frames=frames,
141             runtime_sec=runtime
142         )
143
144     def plot_convergence(history_best: List[float], title: str, out_path:
       str):
145         plt.figure(figsize=(6, 4))
146         if len(history_best) == 0:
147             history_best = [np.nan]
148         xs = np.arange(1, len(history_best) + 1)
149         plt.semilogy(xs, history_best)
150         plt.xlabel("generation")
151         plt.ylabel("best f(x) (log)")
152         plt.title(title)
153         plt.grid(True, which="both", ls=":")
154         os.makedirs(os.path.dirname(out_path), exist_ok=True)
155         plt.tight_layout()
156         plt.savefig(out_path, dpi=120)
157         plt.close()
158
159     def plot_surface_2d(out_path: str):
160         grid_n = 200
161         xs = np.linspace(BOUND_LOW, BOUND_HIGH, grid_n)
162         ys = np.linspace(BOUND_LOW, BOUND_HIGH, grid_n)
163         Xg, Yg = np.meshgrid(xs, ys)
164         Z = Xg**2 + Yg**2
165         fig = plt.figure(figsize=(7, 6))
166         ax = fig.add_subplot(111, projection="3d")
167         ax.plot_surface(Xg, Yg, Z, linewidth=0, antialiased=True, alpha=0.9)
168         ax.set_title("De Jong f1 surface (n=2)")
169         ax.set_xlabel("x1")

```

```

170     ax.set_ylabel("x2")
171     ax.set_zlabel("f(x)")
172     os.makedirs(os.path.dirname(out_path), exist_ok=True)
173     plt.tight_layout()
174     plt.savefig(out_path, dpi=120)
175     plt.close()
176
177 def plot_isosurfaces_n3(points: np.ndarray, out_path: str):
178     fig = plt.figure(figsize=(7, 6))
179     ax = fig.add_subplot(111, projection="3d")
180     ax.scatter(points[:, 0], points[:, 1], points[:, 2], s=10,
181               depthshade=True)
182     radii = [0.5, 1.0, 2.0, 3.0]
183     phi = np.linspace(0, np.pi, 40)
184     th = np.linspace(0, 2*np.pi, 40)
185     PHI, TH = np.meshgrid(phi, th)
186     for r in radii:
187         Xs = r * np.sin(PHI) * np.cos(TH)
188         Ys = r * np.sin(PHI) * np.sin(TH)
189         Zs = r * np.cos(PHI)
190         ax.plot_wireframe(Xs, Ys, Zs, rstride=4, cstride=4, alpha=0.25)
191     ax.set_title("n=3 population and isosurfaces f(x)=const")
192     ax.set_xlabel("x1")
193     ax.set_ylabel("x2")
194     ax.set_zlabel("x3")
195     ax.set_xlim(BOUND_LOW, BOUND_HIGH)
196     ax.set_ylim(BOUND_LOW, BOUND_HIGH)
197     ax.set_zlim(BOUND_LOW, BOUND_HIGH)
198     os.makedirs(os.path.dirname(out_path), exist_ok=True)
199     plt.tight_layout()
200     plt.savefig(out_path, dpi=120)
201     plt.close()
202
203 def gens_to_threshold(history: List[float], thr: float) -> int:
204     for i, v in enumerate(history, start=1):
205         if v <= thr:
206             return i
207     return len(history)
208
209 if __name__ == "__main__":
210     os.makedirs("outputs", exist_ok=True)
211
212     cfg_n2 = ESConfig(
213         n_dim=2, mu=15, lam=90, init_step=1.5,
214         max_generations=500, target_f=1e-6, seed=123,
215         adapt_scale=0.25, frame_stride=20, save_frames=True,
216         run_name="n2_mu15_lam90"
217     )
218     res_n2 = run_es(cfg_n2)
219     plot_convergence(res_n2.history_best, "Convergence n=2 (mu/lam
220                       =15/90)", "outputs/n2_convergence.png")
221     plot_surface_2d("outputs/n2_surface.png")
222
223     cfg_n3 = ESConfig(
224         n_dim=3, mu=15, lam=90, init_step=1.5,
225         max_generations=700, target_f=1e-6, seed=321,
226         adapt_scale=0.25, frame_stride=20, save_frames=False,
227         run_name="n3_mu15_lam90"
228     )
229     res_n3 = run_es(cfg_n3)

```

```

229 plot_convergence(res_n3.history_best, "Convergence n=3 ( $\mu/\lambda$ 
    =15/90)", "outputs/n3_convergence.png")
230
231 pop_settings = [(10, 50), (20, 100), (40, 200)]
232 rows_n2, rows_n3 = [], []
233 for mu, lam in pop_settings:
234     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)
235     r_a = run_es(cfg_a)
236     rows_n2.append({
237         "mu/lam": f"{mu}/{lam}",
238         "final_f": r_a.best_f,
239         "gens_to_f_lt_0.01": gens_to_threshold(r_a.history_best, 1e
            -2),
240         "time_sec": r_a.runtime_sec
241     })
242     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)
243     r_b = run_es(cfg_b)
244     rows_n3.append({
245         "mu/lam": f"{mu}/{lam}",
246         "final_f": r_b.best_f,
247         "gens_to_f_lt_0.01": gens_to_threshold(r_b.history_best, 1e
            -2),
248         "time_sec": r_b.runtime_sec
249     })
250 df_pop_n2 = pd.DataFrame(rows_n2)
251 df_pop_n3 = pd.DataFrame(rows_n3)
252 df_pop_n2.to_csv("outputs/table_population_n2.csv", index=False)
253 df_pop_n3.to_csv("outputs/table_population_n3.csv", index=False)
254
255 step_settings = [0.05, 0.1, 0.5, 1.0, 2.0]
256 rows_s2, rows_s3 = [], []
257 for s0 in step_settings:
258     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)
259     r_a = run_es(cfg_a)
260     rows_s2.append({
261         "init_step": s0,
262         "final_f": r_a.best_f,
263         "gens_to_f_lt_0.01": gens_to_threshold(r_a.history_best, 1e
            -2),
264         "time_sec": r_a.runtime_sec
265     })
266     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)
267     r_b = run_es(cfg_b)
268     rows_s3.append({
269         "init_step": s0,
270         "final_f": r_b.best_f,
271         "gens_to_f_lt_0.01": gens_to_threshold(r_b.history_best, 1e
            -2),
272         "time_sec": r_b.runtime_sec
273     })
274 df_step_n2 = pd.DataFrame(rows_s2)
275 df_step_n3 = pd.DataFrame(rows_s3)
276 df_step_n2.to_csv("outputs/table_step_n2.csv", index=False)

```

```

277 df_step_n3.to_csv("outputs/table_step_n3.csv", index=False)
278
279 if len(res_n2.frames) >= 1:
280     last_pop_img = res_n2.frames[-1]
281 else:
282     last_pop_img = ""
283
284 if res_n3.generations > 0:
285     rng_tmp = np.random.default_rng(999)
286     pop3 = rng_tmp.uniform(BOUND_LOW, BOUND_HIGH, size=(300, 3))
287     plot_isosurfaces_n3(pop3, "outputs/n3_population_isosurfaces.png")
288
289 print("n=2 result:")
290 print(f"x* = {np.array2string(res_n2.best_x, precision=6, separator
291     =', ')}")
292 print(f"f(x*) = {res_n2.best_f:.6e}")
293 print(f"generations = {res_n2.generations}")
294 print(f"time_sec = {res_n2.runtime_sec:.4f}")
295 print(f"convergence_plot = outputs/n2_convergence.png")
296 print(f"surface_plot = outputs/n2_surface.png")
297 if last_pop_img:
298     print(f"frames example = {last_pop_img}")
299
300 print("\nn=3 result:")
301 print(f"x* = {np.array2string(res_n3.best_x, precision=6, separator
302     =', ')}")
303 print(f"f(x*) = {res_n3.best_f:.6e}")
304 print(f"generations = {res_n3.generations}")
305 print(f"time_sec = {res_n3.runtime_sec:.4f}")
306 print(f"convergence_plot = outputs/n3_convergence.png")
307 print(f"3D volume plot (isosurfaces) = outputs/
308     n3_population_isosurfaces.png")
309
310 print("\nTables:")
311 print("outputs/table_population_n2.csv")
312 print("outputs/table_population_n3.csv")
313 print("outputs/table_step_n2.csv")
314 print("outputs/table_step_n3.csv")
315
316 print("\nMain required figures (4):")
317 print("1) outputs/n2_surface.png")
318 print("2) outputs/n2_convergence.png")
319 print("3) outputs/n3_population_isosurfaces.png")
320 print("4) outputs/n3_convergence.png")

```