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»

«Лабораторная работа №1»

Простой генетический алгоритм
Вариант №17

Студент: _____ Салимли Айзек Мухтар Оглы

Преподаватель: _____ Большаков Александр Афанасьевич

«____»_____ 20__ г.

Санкт-Петербург, 2025

Содержание

Введение **3**

1 Постановка задачи **4**

2 Определения **5**

3 Генетические операторы **6**
3.1 Оператор репродукции 6
3.2 Оператор скрещивания (кроссинговер) 6
3.3 Оператор мутации 6

4 Программная реализация **7**

5 Результаты **9**

6 Выводы и исследование **11**

7 Заключение **14**

8 Ответ на контрольный вопрос **15**

Список литературы **16**

Приложение А **17**

Введение

Генетические алгоритмы (ГА) используют принципы и терминологию, заимствованные у биологической науки генетики. В ГА каждая особь представляет потенциальное решение некоторой проблемы. В классическом ГА особь кодируется строкой двоичных символов хромосомой, каждый бит которой называется геном. Множество особей потенциальных решений составляет популяцию. Поиск (суб)оптимального решения проблемы выполняется в процессе эволюции популяции - последовательного преобразования одного конечного множества решений в другое с помощью генетических операторов репродукции, кроссовера и мутации.

Предварительно простой ГА случайным образом генерирует начальную популяцию строк (хромосом). Затем алгоритм генерирует следующее поколение (популяцию), с помощью трех основных генетических операторов:

1. Оператор репродукции (ОР);
2. Оператор скрещивания (кроссовера, ОК);
3. Оператор мутации (ОМ).

ГА работает до тех пор, пока не будет выполнено заданное количество поколений (итераций) процесса эволюции или на некоторой генерации будет получено заданное качество или вследствие преждевременной сходимости при попадании в некоторый локальный оптимум. На Рис. 1 представлен простой генетический алгоритм.

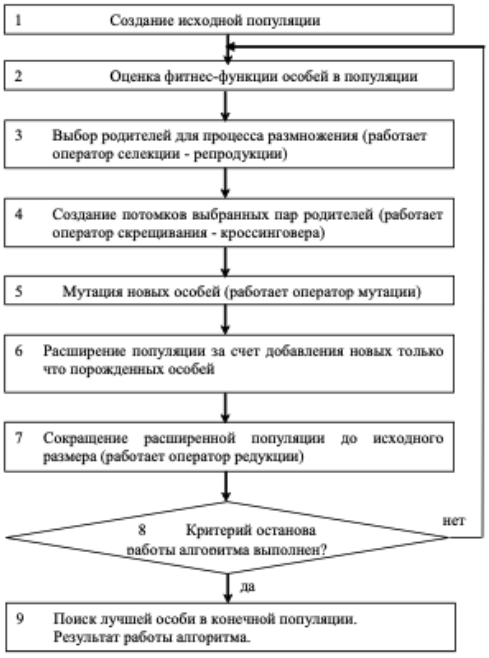


Рис. 1: Простой генетический алгоритм

1 Постановка задачи

- Изучить теоритический материал;
- Выбрать индивидуальный вариант задания;
- Рассмотреть способы выполнения операторов репродукции, скрещивания и мутации;
- Реализовать программу, на выбранном языке программирования с комментариями и выводами.

Индивидуальное задание: №17
Дано: Функция одной переменной $f(x) = \frac{\cos(\exp(x))}{\sin(\ln(x))}$, промежуток нахождения решения: $x \in [2, 4]$.

Требуется:

1. Разобрать простой генетический алгоритм для нахождения минимума функции на промежутке $[2, 4]$;
2. Исследовать зависимость времени поиска, числа поколений и точности нахождения решения от числа особей в популяции и вероятность кроссинговера и мутации;
3. Визуализировать результаты, графиками функции с указанием найденного экстремума для каждого поколения;
4. Сравнить полученные результаты с действительными значениями.

Ограничения:

1. $min\ \ f(x) = \frac{\cos(\exp(x))}{\sin(\ln(x))} \approx -1.3847$;
2. точность: три знака после запятой;

2 Определения

Ген - элементарный код в хромосоме s_i, называемый также признаком или детектором (в классическом ГА $s_i = 0, 1$).

Хромосома - упорядоченная последовательность генов в виде закодированной структуры данных $S = (s_1, s_2, \ldots, s_n)$, определяющая решение (в простейшем случае двоичная последовательность строка, где $s_i = 0, 1$).

Локус - местоположение (позиция, номер бита) данного гена в хромосоме.

Аллель - значение, которое принимает данный ген (например, 0 или 1).

Особь - одно потенциальное решение задачи (представляемое хромосомой).

Популяция - множество особей (хромосом), представляющих потенциальные решения.

Поколение - текущая популяция ГА на данной итерации алгоритма.

Генотип - набор хромосом данной особи. В популяции могут использоваться как отдельные хромосомы, так и целые генотипы.

Генофонд - множество всех возможных генотипов.

Фенотип - набор значений, соответствующий данному генотипу. Это декодированное множество параметров задачи (например, десятичное значение x, соответствующее двоичному коду).

Размер популяции - число особей в популяции.

Число поколений - количество итераций, в течение которых производится поиск.

Селекция - совокупность правил, определяющих выживание особей на основе значений целевой функции.

Эволюция популяции - чередование поколений, в которых хромосомы изменяют свои признаки, чтобы каждая новая популяция лучше приспосабливалась к среде.

Фитнесс-функция - функция полезности, определяющая меру приспособленности особи. В задачах оптимизации она совпадает с целевой функцией или описывает близость к оптимальному решению.

3 Генетические операторы

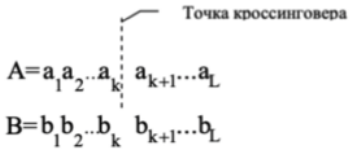
3.1 Оператор репродукции

Оператор репродукции - процесс копирования хромосом в промежуточную популяцию для дальнейшего размножения в соответствии со значениями фитнесс-функции. В данной работе рассматривается метод колеса рулетки. Каждой хромосоме соответствует сектор, пропорциональный значению фитнесс-функции. Хромосомы с большим значением имеют больше шансов попасть в следующее поколение.

3.2 Оператор скрещивания (кроссинговер)

Одноточечный кроссинговер выполняется следующим образом:

1. Из промежуточной популяции выбираются две хромосомы (родители).



2. Определяется случайная точка скрещивания $k \in [1, n-1]$, где n - длина хромосомы.

$$A' = a_1 a_2 .. a_k \;\; b_{k+1} ... b_L$$

$$B' = b_1 b_2 .. b_k \;\; a_{k+1} ... a_L$$

3. Две новые хромосомы (потомки) формируются путем обмена подстрок после точки k.

3.3 Оператор мутации

Мутация применяется с малой вероятностью $P_M \approx 0.001$:

1. В хромосоме $A = a_1 a_2 \dots a_n$ выбирается случайная позиция k.
2. a_k-й ген хромосомы инвертируется $\Rightarrow a'_k = \overline{a_k}$.

4 Программная реализация

В рамках лабораторной работы был реализован ноутбук **GA_Lab_1.ipynb**, в котором был реализован простой генетический алгоритм для нахождения минимума функции на промежутке $[2, 4]$.

- Редактором кода был выбран **VS Code**;
- Для реализации генетического алгоритма был выбран язык программирования **Python 3.13.5**;
- Для структурирования кода, был использован **Jupyter Notebook**;
- В качестве библиотеки для визуализации была выбрана библиотека **Matplotlib**.

Цель. Найти минимум функции:

$$f(x) = \frac{\cos(e^x)}{\sin(\ln x)}, \quad x \in [2, 4].$$

Подход. Простой генетический алгоритм (ГА) над *хромосомами* (каждая особь — это число x), с турнирной селекцией, арифметическим кроссовером, мутацией, элитизмом.

Визуализация. Печать таблицы по поколениям и графические «снимки» эволюции популяции поверх кривой целевой функции.

Ключевые компоненты

Целевая функция $f(x)$

```
def f(x):
    try:
        return cos(exp(x)) / sin(log(x))
    except:
        return inf
```

При проблемах области определения (например, деление на почти ноль) возвращает $+\infty$, чтобы такие точки автоматически «проигрывали» при минимизации.

Конфигурация GAConfig

- Диапазон поиска: $x_{\min} = 2.0$, $x_{\max} = 4.0$, округление до `precision=3`.
- Параметры ГА: размер популяции, число поколений, вероятности кроссовера и мутации, размер турнира, элитизм.
- Гауссова мутация с начальными/минимальными σ.
- Параметр `cx_alpha` расширяет отрезок между родителями при арифметическом кроссовере.

Инициализация популяции `init_population`

Случайные $x \in [x_{\min}, x_{\max}]$ с округлением до `precision`.

Как работает ГА в коде

1. **Оценка приспособленности**

```
fitness = evaluate(pop, f)
```

Для минимизации используется прямое сравнение значений $f(x)$ — меньше значит лучше. Значение `inf` отталкивает невалидные точки.

2. **Селекция — турнирная**

```
def tournament_select(pop, fitness, k):
    # выбираем k случайных индексов, берем лучшего (min fitness)
```

Параметр `tournament_size` управляет «давлением отбора»: чем больше k, тем сильнее отбор.

3. **Кроссовер — арифметический**

```
c1, c2 = arithmetic_crossover(p1, p2, alpha=cfg.cx_alpha)
```

Пусть $L = |p_1 - p_2|$. Потомки берутся из расширенного отрезка $[p_{\min} - \alpha L,\ p_{\max} + \alpha L]$, что даёт больше разнообразия.

4. **Мутация**

```
sigma = max(sigma_floor, sigma_init * sigma_decay**gen)
x' = x + Normal(0, sigma)
```

В начале σ больше (широкий поиск), затем затухает (локальное уточнение). Результаты ограничиваются диапазоном и округляются.

5. **Элитизм**

```
elite_idx = argsort(fitness)[:elitism]
next_pop = [pop[i] for i in elite_idx]
```

Лучшая часть популяции без изменений переходит в следующее поколение.

6. **Формирование нового поколения** После элитизма добор: турнир → кроссовер (с вероятностью p_c) → мутация → добавление. В конце все значения округляются и оцениваются заново.

7. **История и останов** Хранится история популяций (`pop_history`), лучшие пары $(x, f(x))$, таблица метрик. Критерий остановки — фиксированное число поколений.

Вспомогательные части

- **Табличный лог** `print_table` — выводит `best_x`, `best_f`, `mean_f` по поколениям.
- **Визуализация** `plot_snapshot` — график функции и положение популяции, текущего и предыдущего лучших.
- **Детерминизм.** `random.seed(cfg.seed)` обеспечивает повторяемость экспериментов.

5 Результаты

Ниже, на Рис. 2 - 5 представлены результаты работы генетического алгоритма.

- $N = 20$ — размер популяции;
- $p_c = 0.85$ — вероятность кроссинговера;
- $p_m = 0.12$ — вероятность мутации;
- поиск на отрезке $[2, 4]$ с округлением до 10^{-3};

С каждым поколением точность найденного минимума возрастает: — средняя приспособленность популяции снижается, — лучшая особь стабильно улучшается.

На ранних поколениях (например, $g = 0, 5$) особи покрывают значительную часть области поиска; к эволюции ($g \approx 10$) популяция «конденсируется» в окрестности экстремума; к финалу ($g = 15$) достигается заданная точность (округление 10^{-3}), а лучшая особь находится вблизи минимума.

Визуализация включает по-поколенческие снимки: график функции, текущую популяцию, лучшую особь текущего поколения (звёздочка) и лучшую особь предыдущего поколения (для сравнения). Алгоритм завершает работу по фиксированному числу поколений (15); помимо графиков печатается таблица с метриками по поколениям: `best_x`, `best_f`, `mean_f`.

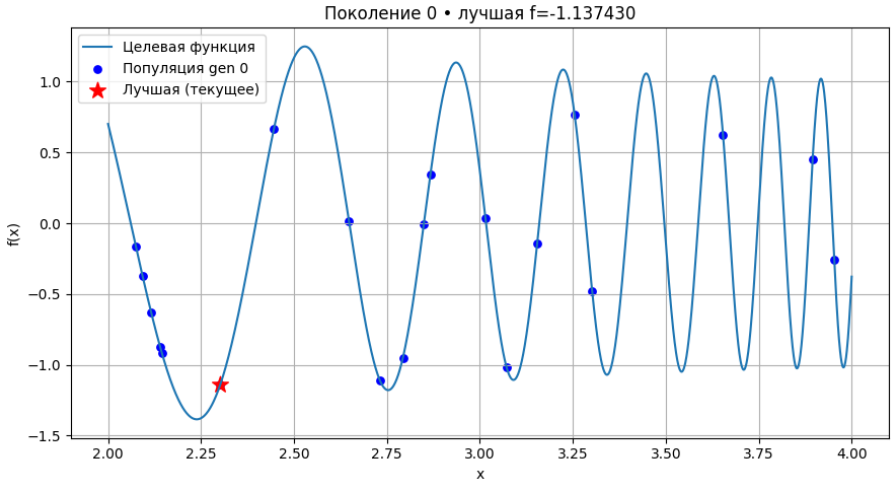


Рис. 2: Поколение 0 - начальная популяция

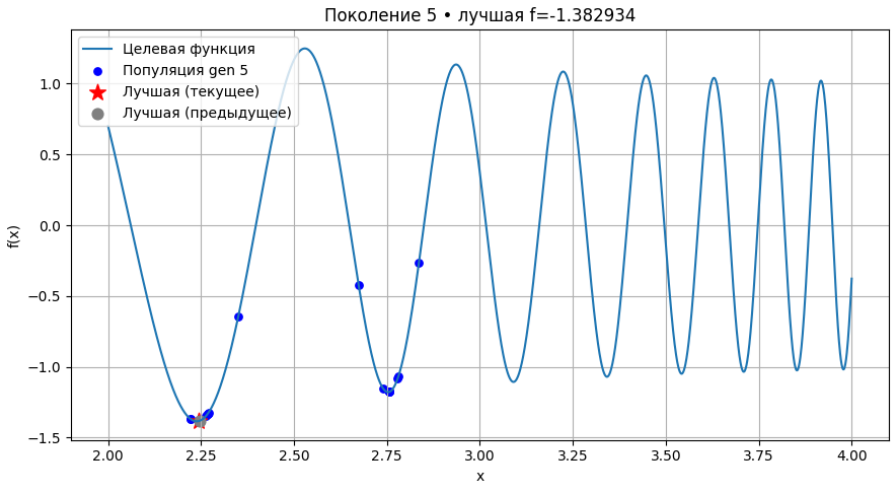


Рис. 3: Поколение 5

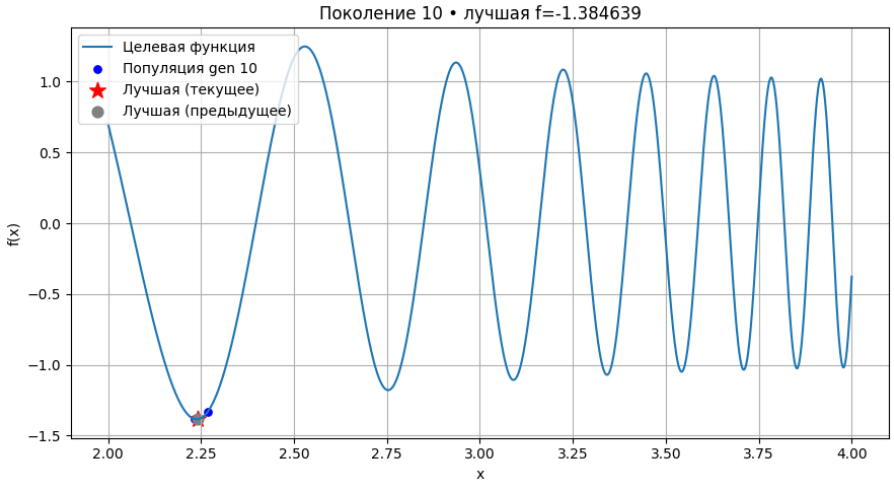


Рис. 4: Поколение 10 - концентрация популяции

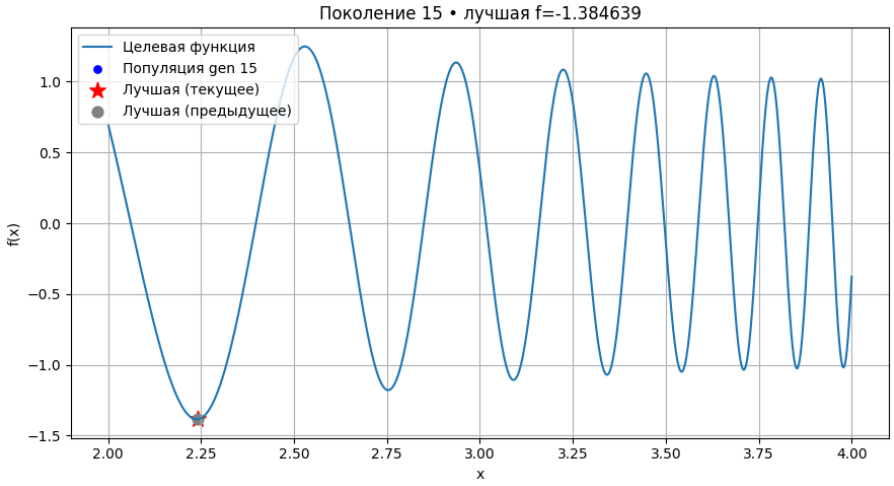


Рис. 5: Финальное поколение 15

6 Выводы и исследование

Анализ результатов работы

В эксперименте использовались следующие параметры генетического алгоритма:

$$N = 20, \qquad p_c = 0.85, \qquad p_m = 0.12.$$

Алгоритм находит минимум функции

$$f(x) = \frac{\cos(e^x)}{\sin(\ln x)}, \quad x \in [2, 4]$$

достаточно быстро, и результаты близки к тем, что получаются при плотном сканировании (reference: $x \approx 2.2385,\ f \approx -1.38478$).

Динамика сходимости. Уже к поколению $g = 10$ лучшая особь демонстрирует значение минимума; к $g = 15$ популяция практически полностью концентрируется в его окрестности. Динамика значений лучшего и среднего по популяции представлена в таблице 1.

Таблица 1: Эволюция лучшей и средней приспособленности по поколениям

Gen	Best_x	Best_f	Mean_f	Time(s)
0	2.302	-1.137430	-0.308400	0.0006
1	2.757	-1.176649	-0.856102	0.0001
2	2.756	-1.177595	-0.263769	0.0001
3	2.268	-1.331403	-1.160674	0.0001
4	2.246	-1.381351	-1.144883	0.0001
5	2.244	-1.382934	-1.358064	0.0002
6	2.240	-1.384639	-1.382867	0.0001
7	2.240	-1.384639	-1.383304	0.0001
8	2.240	-1.384639	-1.384462	0.0003
9	2.240	-1.384639	-1.382024	0.0001
10	2.240	-1.384639	-1.384639	0.0001
11	2.240	-1.384639	-1.384639	0.0001
12	2.240	-1.384639	-1.382654	0.0001
13	2.240	-1.384639	-1.371588	0.0001
14	2.240	-1.384639	-1.384639	0.0001

В данном запуске использован размер популяции $N = 50$. Таблица ниже показывает динамику лучшего решения и среднего значения по популяции по поколениям.

Таблица 2: Эволюция лучшего и среднего значения целевой функции ($N = 50$)

Gen	Best_x	Best_f	Mean_f	Time(s)
0	2.745	-1.168921	-0.645240	0.0003
1	2.223	-1.370500	-0.748739	0.0002
2	2.237	-1.384646	-1.189021	0.0002
3	2.238	-1.384764	-1.352526	0.0003
4	2.238	-1.384764	-1.350280	0.0005
5	2.238	-1.384764	-1.361282	0.0002
6	2.238	-1.384764	-1.370731	0.0002
7	2.238	-1.384764	-1.377510	0.0002
8	2.238	-1.384764	-1.384757	0.0002
9	2.238	-1.384764	-1.381099	0.0002
10	2.238	-1.384764	-1.379964	0.0002
11	2.238	-1.384764	-1.371425	0.0008
12	2.238	-1.384764	-1.384676	0.0004
13	2.238	-1.384764	-1.381669	0.0003
14	2.238	-1.384764	-1.384630	0.0002

Результаты популяции $N = 100$

В данном запуске использован размер популяции $N = 100$. Уже к поколениям $g = 10$–15 лучшая особь достигает значения минимума. Однако к $g = 15$ популяция ещё *не полностью сконцентрировалась* в его окрестности: среднее значение по популяции остаётся заметно выше по модулю минимального.

Таблица 3: Эволюция лучшего и среднего значения целевой функции ($N = 100$)

Gen	Best_x	Best_f	Mean_f	Time(s)
0	2.745	-1.168921	-0.409121	0.0008
1	2.248	-1.379278	-0.548675	0.0013
2	2.245	-1.382203	-0.759144	0.0004
3	2.240	-1.384639	-1.211791	0.0010
4	2.238	-1.384764	-1.373112	0.0003
5	2.238	-1.384764	-1.361731	0.0004
6	2.238	-1.384764	-1.377298	0.0003
7	2.238	-1.384764	-1.369395	0.0003
8	2.238	-1.384764	-1.373058	0.0011
9	2.238	-1.384764	-1.379324	0.0008
10	2.238	-1.384764	-1.351899	0.0008
11	2.238	-1.384764	-1.383980	0.0003
12	2.238	-1.384764	-1.383903	0.0003
13	2.238	-1.384764	-1.381956	0.0003
14	2.238	-1.384764	-1.381509	0.0011

Исследование влияния параметров. Для исследования влияния размера популяции и параметров генетического алгоритма были проведены эксперименты с различными конфигурациями:

- **Размер популяции:** $N \in \{20, 50, 100\}$
- **Вероятность кроссовера:** $p_c \in \{0.3, 0.5, 0.6, 0.85\}$
- **Вероятность мутации:** $p_m \in \{0.01, 0.05, 0.12, 0.2\}$

- **Число поколений:** 15 (для сравнения результатов)

Результаты для $N = 20$: При малых популяциях повышение p_m ускоряет поиск; наилучшее время при $p_c = 0.5$, $p_m = 0.12$ (сходимость к поколению 8-10). Для стабильной работы оптимально $p_c = 0.85$, $p_m = 0.12$.

Результаты для $N = 50$: при умеренной мутации $p_m \in [0.05, 0.12]$; лучшая сходимость при $p_c = 0.85$, $p_m = 0.05$ (сходимость к поколению 4-6). Слишком большая мутация ($p_m = 0.2$) резко ухудшает результаты.

Результаты для $N = 100$: Оптимальны низкие p_m; лучший результат при $p_c = 0.85$, $p_m = 0.05$ (сходимость к поколению 3-5). При $p_m = 0.2$ решение часто не находится за 15 поколений.

Влияние размера популяции: Суммарное время часто растёт из-за большей стоимости одной итерации. При $N = 100$ достигается быстрая сходимость (3-5 поколений), но вычислительная сложность возрастает.

Практические выводы:

- Для умеренных затрат времени и стабильной сходимости разумно выбирать $N \approx 50$, $p_c \approx 0.85$, $p_m \approx 0.05$.
- Оптимальное p_m снижается с ростом N: при малых популяциях полезна более агрессивная мутация, при больших — слабая.
- Слишком большие значения p_m и p_c могут разрушать хорошие решения и ухудшать сходимость; стоит избегать $p_m \geq 0.2$ и высоких p_c при больших N.
- Минимум достигается уже к поколениям $3 \div 6$, после чего изменения очень малые.
- Лучшее значение $f \approx -1.38476$ близко к сканковому -1.38478, погрешность порядка 10^{-4}.
- Среднее значение приспособленности популяции стабильно улучшается, что говорит о эффективной работе генетических операторов.

7 Заключение

В ходе первой лабораторной работы:

1. Был изучен теоретический материал, основная терминология ГА, генетические операторы, использующиеся в простых ГА;
2. Реализована программа на языке Python для нахождения минимума заданной функции;
3. Проведено исследование зависимости поколения от мощности популяции и коэффициентов кроссовера и мутации.

В результате проведённых экспериментов было установлено, что генетический алгоритм эффективно находит минимум функции $f(x) = \frac{\cos(e^x)}{\sin(\ln x)}$ на отрезке $[2, 4]$ с высокой точностью.

Оптимальные параметры для данной задачи: размер популяции $N \in \{20, 50\}$, вероятность кроссовера $p_c = 0.85$, вероятность мутации $p_m = 0.05$.

8 Ответ на контрольный вопрос

Вопрос: Какие генетические операторы используются в ГА?
Ответ: В генетическом алгоритме используются три основных оператора:

1. **Оператор репродукции (селекции)** — выбирает наиболее приспособленных особей для дальнейшего размножения. Обеспечивает сохранение лучших решений и направляет эволюцию в сторону улучшения качества популяции.
2. **Оператор скрещивания (кроссовера)** — создает новых потомков путем комбинирования генетического материала двух родительских особей. Позволяет исследовать новые области пространства решений и объединять полезные признаки от разных особей.
3. **Оператор мутации** — случайным образом изменяет отдельные гены в хромосомах. Предотвращает преждевременную сходимость алгоритма, обеспечивает генетическое разнообразие популяции и позволяет исследовать новые области поиска.

Список литературы

1. Методические указания по выполнению лабораторных работ к курсу "Генетические алгоритмы 119 стр.

Приложение А

Листинг 1: Приложение А: исходный код

```
import math
import random
from dataclasses import dataclass
from typing import Callable, List, Tuple, Dict
import numpy as np
import matplotlib.pyplot as plt


def f(x: float) -> float:
    try:
        return math.cos(math.exp(x)) / math.sin(math.log(x))
    except Exception:
        return float("inf")

@dataclass
class GAConfig:
    x_min: float = 2.0
    x_max: float = 4.0
    precision: int = 3

    pop_size: int = 50
    generations: int = 60

    crossover_rate: float = 0.85
    mutation_rate: float = 0.12
    elitism: int = 4
    tournament_size: int = 4

    sigma_init: float = 0.08
    sigma_floor: float = 0.001
    sigma_decay: float = 0.96

    cx_alpha: float = 0.10

    seed: int = 7

    avoid_min_at_start: bool = True
    avoid_window_dx: float = 0.06

def clamp(x, a, b): return max(a, min(b, x))
def roundp(x, p): return round(x, p)

def arithmetic_crossover(p1: float, p2: float, alpha: float) -> Tuple[float,
    float]:
    lo, hi = (p1, p2) if p1 <= p2 else (p2, p1)
    L = hi - lo
    a = lo - alpha * L
    b = hi + alpha * L
    r = random.random()
    c1 = a + r * (b - a)
    c2 = a + (1 - r) * (b - a)
    return c1, c2

def mutate(x: float, rate: float, sigma: float, cfg: GAConfig) -> float:
    if random.random() < rate:
        x = x + random.gauss(0.0, sigma)
```

```
        x = clamp(x, cfg.x_min, cfg.x_max)
        x = roundp(x, cfg.precision)
    return x

def evaluate(pop: List[float], func: Callable[[float], float]) -> List[float]:
    return [func(x) for x in pop]

def tournament_select(pop: List[float], fitness: List[float], k: int) -> float:
    idxs = random.sample(range(len(pop)), k)
    best_idx = min(idxs, key=lambda i: fitness[i])
    return pop[best_idx]

def dense_scan_min(x_min=2.0, x_max=4.0) -> Tuple[float, float]:
    xs = np.linspace(x_min, x_max, 40001)
    ys = np.cos(np.exp(xs)) / np.sin(np.log(xs))
    i = int(np.argmin(ys))
    return float(xs[i]), float(ys[i])

def init_population(cfg: GAConfig, forbid: Tuple[float, float] | None) -> List[float]:
    pop = []
    while len(pop) < cfg.pop_size:
        x = roundp(random.uniform(cfg.x_min, cfg.x_max), cfg.precision)
        if forbid and (forbid[0] <= x <= forbid[1]):
            continue
        pop.append(x)
    return pop

def run_ga(func: Callable[[float], float], cfg: GAConfig) -> Dict:
    random.seed(cfg.seed)

    approx_x, approx_y = dense_scan_min(cfg.x_min, cfg.x_max)
    forbid = None
    if cfg.avoid_min_at_start:
        a = clamp(approx_x - cfg.avoid_window_dx, cfg.x_min, cfg.x_max)
        b = clamp(approx_x + cfg.avoid_window_dx, cfg.x_min, cfg.x_max)
        forbid = (a, b)

    pop = init_population(cfg, forbid)
    fitness = evaluate(pop, func)

    pop_history: List[List[float]] = [pop.copy()]
    best_history: List[Tuple[float, float]] = []
    table_rows: List[Dict] = []

    for gen in range(cfg.generations):
        i_best = int(np.argmin(fitness))
        best_x_gen = pop[i_best]
        best_f_gen = float(fitness[i_best])

        best_history.append((best_x_gen, best_f_gen))
        table_rows.append({
            "generation": gen,
            "best_x": best_x_gen,
            "best_f": best_f_gen,
            "mean_f": float(np.mean(fitness)),
        })

        elite_idx = np.argsort(fitness)[:cfg.elitism]
```

```
        next_pop = [pop[i] for i in elite_idx]

        sigma = max(cfg.sigma_floor, cfg.sigma_init * (cfg.sigma_decay ** gen))

        while len(next_pop) < cfg.pop_size:
            p1 = tournament_select(pop, fitness, cfg.tournament_size)
            p2 = tournament_select(pop, fitness, cfg.tournament_size)

            if random.random() < cfg.crossover_rate:
                c1, c2 = arithmetic_crossover(p1, p2, cfg.cx_alpha)
            else:
                c1, c2 = p1, p2

            c1 = mutate(c1, cfg.mutation_rate, sigma, cfg)
            c2 = mutate(c2, cfg.mutation_rate, sigma, cfg)

            next_pop.extend([c1, c2])

        pop = [roundp(clamp(x, cfg.x_min, cfg.x_max), cfg.precision) for x in next_pop[:cfg.pop_size]]
        fitness = evaluate(pop, func)
        pop_history.append(pop.copy())

    final_best = best_history[-1]
    return {
        "pop_history": pop_history,
        "best_history": best_history,
        "final_best": final_best,
        "table": table_rows,
        "approx_min": (approx_x, approx_y),
        "cfg": cfg,
    }

def compute_curve(x_min: float, x_max: float):
    xs = np.linspace(x_min, x_max, 2000)
    ys = np.cos(np.exp(xs)) / np.sin(np.log(xs))
    return xs, ys

def plot_snapshot(xs, ys, gen_idx: int, pop_history: List[List[float]], best_history: List[Tuple[float, float]]):
    pop_cur = pop_history[gen_idx]
    best_cur_x, best_cur_f = best_history[gen_idx] if gen_idx < len(best_history) else best_history[-1]
    prev_best = best_history[gen_idx - 1] if gen_idx - 1 >= 0 else None

    plt.figure(figsize=(9, 5))
    plt.plot(xs, ys, label="Target function")
    plt.scatter(pop_cur, [f(x) for x in pop_cur], s=28, label=f"Population gen {gen_idx}", color="blue")
    plt.scatter([best_cur_x], [best_cur_f], s=140, marker="*", color="red", label="Best (cur)")
    if prev_best is not None:
        plt.scatter([prev_best[0]], [prev_best[1]], s=60, color="gray", label="Best (prev)")
    plt.title(f"Generation {gen_idx} best f={best_cur_f:.6f}")
    plt.xlabel("x"); plt.ylabel("f(x)")
    plt.grid(True); plt.legend()
    plt.tight_layout()
    plt.show()

```

```python
def print_table(rows: List[Dict]) -> None:
    header = f"{'Gen':>4} | {'Best_x':>8} | {'Best_f':>12} | {'Mean_f':>12}"
    print("\n" + header)
    print("-" * len(header))
    for r in rows:
        print(f"{r['generation']:>4} | {r['best_x']:>8.3f} | {r['best_f']:>12.6f} | {r['mean_f']:>12.6f}")

def main():
    cfg = GAConfig(
        pop_size=100,
        generations=15,
        crossover_rate=0.85,
        mutation_rate=0.05,
        elitism=4,
        tournament_size=4,
        precision=3,
        sigma_init=0.08,
        sigma_floor=0.001,
        sigma_decay=0.96,
        cx_alpha=0.10,
        seed=7,
        avoid_min_at_start=True,
        avoid_window_dx=0.06,
    )

    res = run_ga(f, cfg)
    pop_history = res["pop_history"]
    best_history = res["best_history"]
    rows = res["table"]
    approx_x, approx_y = res["approx_min"]

    print_table(rows)

    xs, ys = compute_curve(cfg.x_min, cfg.x_max)

    gens_to_show = [0, 5, 10, 20, 30, 40, 50, 60]
    gens_to_show = [g for g in gens_to_show if 0 <= g < len(pop_history)]
    if (len(pop_history)-1) not in gens_to_show:
        gens_to_show.append(len(pop_history)-1)

    for g in gens_to_show:
        plot_snapshot(xs, ys, g, pop_history, best_history)


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