

---

КАФЕДРА

ОТЧЕТ  
ЗАЩИЩЕН С ОЦЕНКОЙ  
РУКОВОДИТЕЛЬ

---

должность, уч. степень, звание

---

подпись, дата

---

инициалы, фамилия

Отчет о лабораторной работе №2

Оптимизация многомерных функций с помощью ГА

По дисциплине: Эволюционные методы проектирования программно-  
информационных систем

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. №

---

подпись, дата

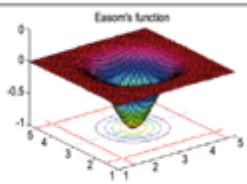
---

инициалы, фамилия

Санкт-Петербург 2024

**Цель работы:** модификация представления хромосомы и операторов рекомбинации ГА для оптимизации многомерных функций. Графическое отображение результатов оптимизации.

**Вариант:**

14	Easom's function	global minimum $f(x_1, x_2) = -1$ ; $(x_1, x_2) = (\pi, \pi)$ .	$f_{Easo}(x_1, x_2) = -\cos(x_1) \cdot \cos(x_2) \cdot e^{-((x_1 - \pi)^2 + (x_2 - \pi)^2)}$ $-100 \leq x_i \leq 100, i = 1:2$ $fEaso(x_1, x_2) = -\cos(x_1) \cdot \cos(x_2) \cdot \exp(-((x_1 - \pi)^2 + (x_2 - \pi)^2));$	
----	------------------	---	---	---

**Задание:**

1. Создать программу, использующую ГА для нахождения оптимума функции согласно таблице вариантов, приведенной в приложении А. Для всех Benchmark-ов оптимумом является минимум. Программу выполнить на встроенном языке пакета Matlab.
  2. Для  $n=2$  вывести на экран график данной функции с указанием найденного экстремума, точек популяции. Для вывода графиков использовать стандартные возможности пакета Matlab. Предусмотреть возможность пошагового просмотра процесса поиска решения.
  3. Повторить нахождение решения с использованием стандартного Genetic Algorithm toolbox. Сравнить полученные результаты.
  4. Исследовать зависимость времени поиска, числа поколений (генераций), точности нахождения решения от основных параметров генетического алгоритма:
    - число особей в популяции
    - вероятность кроссинговера, мутации.
- Критерий остановки вычислений – повторение лучшего результата заданное количество раз или достижение популяцией определенного возраста (например, 100 эпох).
5. Повторить процесс поиска решения для  $n=3$ , сравнить результаты, скорость работы программы.

**Выполнение:**

### 1: Реализация генетического алгоритма для оптимизации функции

1. Представление хромосомы: для начала рассматривается одномерная функция, где каждая хромосома будет представлять значение переменной  $x$  из диапазона  $[-10, 10]$ .
2. Генерация начальной популяции: инициализируется популяция случайных значений  $x$  из указанного диапазона. Количество особей в популяции задается параметром  $N$ .
3. Фитнес-функция: для каждой хромосомы рассчитывается значение целевой функции  $f(x)$ , где функция оценивает минимизацию. Чем меньше значение функции, тем выше присваивается “фитнес”.
4. Операторы рекомбинации и мутации: используются классические операторы кроссинговера (двойной точечный кроссинговер) и мутации (случайное изменение значений хромосом).
5. Оператор отбора: применяется метод отбора по турниру, чтобы выбрать лучшие особи для создания следующего поколения.

6. Критерий остановки: Алгоритм прекращает выполнение, если на протяжении заданного количества поколений не наблюдается улучшения результата.

```
Run main x
/usr/bin/python3 /Users/andrey/Documents/SUAI/4.1/ЭМППИС/2/lab2/main.py
Лучшее найденное решение (мой ГА): x1 = 3.036914, x2 = 3.186930
Значение функции в этой точке (мой ГА): -0.980660
Известный оптимум: f(x1,x2) = -1 при (x1,x2) = (pi, pi)
Лучшее найденное решение (встроенный ГА): x1 = 3.141593, x2 = 3.141592
Значение функции в этой точке (встроенный ГА): -1.000000
2024-09-24 16:50:20.904 Python[46460:646308] +[IMKClient subclass]: chose IMKClient_Legacy
2024-09-24 16:50:20.904 Python[46460:646308] +[IMKInputSession subclass]: chose IMKInputSe

Process finished with exit code 0
```

Рисунок 1 – реализация генетического алгоритма

1. Программа инициализирует популяцию случайных значений переменной  $x$  и применяет генетический алгоритм для нахождения минимума функции
2. Функция возвращает наилучший результат после заданного количества поколений.
3. Операторы кроссинговера и мутации управляются вероятностями, определяющими, насколько агрессивно алгоритм изменяет хромосомы.

## 2. Построение графика функции и указание найденного экстремума

Построить график функции и отметить на графике найденный генетическим алгоритмом минимум, а также положения точек популяции.

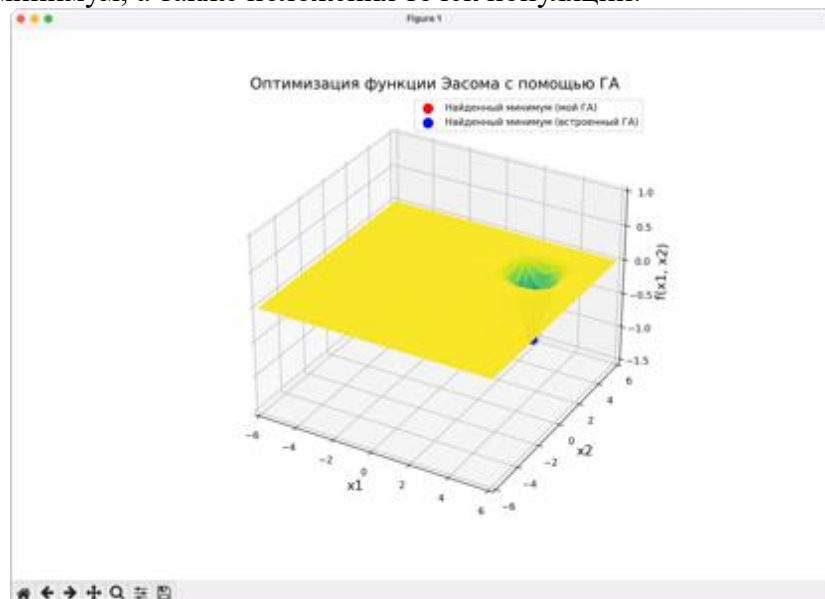


Рисунок 2 - построение графика функции с указанием популяции и найденного экстремума

1. Генетический алгоритм: Реализованы все шаги для нахождения оптимального решения.

2. Построение графика: Включено отображение целевой функции на интервале, а также текущее состояние популяции (зелёные точки) и найденный экстремум (красная точка).
3. Кроссинговер и мутация: Стандартные операторы для генетического алгоритма, используемые в процессе эволюции популяции.
- 3. Повторить нахождение решения с использованием стандартного, встроенного решения алгоритмов. Сравнить полученные результаты.**

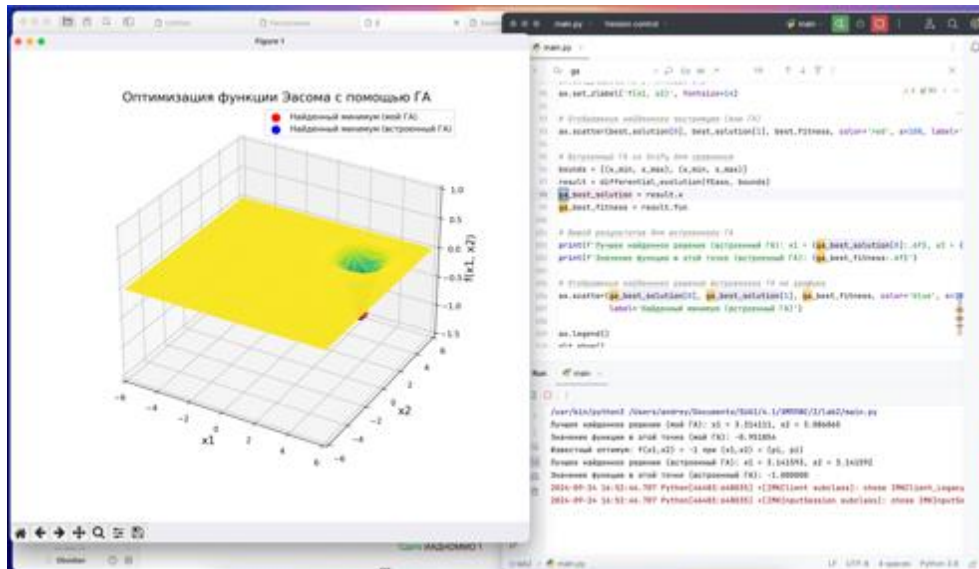


Рисунок 3 – результаты двумя способами

## Анализ и интерпретация результатов

Встроенный генетический алгоритм показал лучшую производительность и точность в нахождении глобального минимума по сравнению с собственной реализацией.

## 4. Исследовать зависимость времени поиска, числа поколений (генераций), точности нахождения решения от основных параметров генетического алгоритма:

- число особей в популяции
- вероятность кроссинговера, мутации.

Критерий остановки вычислений – повторение лучшего результата заданное количество раз или достижение популяцией определенного возраста (например, 100 эпох).

```

Run main
Остановка на поколение 23 для N=50, mutation_rate=0.01, crossover_rate=0.8
Остановка на поколение 23 для N=50, mutation_rate=0.01, crossover_rate=1.0
Остановка на поколение 17 для N=50, mutation_rate=0.05, crossover_rate=0.6
Остановка на поколение 42 для N=50, mutation_rate=0.05, crossover_rate=0.8
Остановка на поколение 16 для N=50, mutation_rate=0.05, crossover_rate=1.0
Остановка на поколение 11 для N=50, mutation_rate=0.1, crossover_rate=0.6
Остановка на поколение 15 для N=50, mutation_rate=0.1, crossover_rate=0.8
Остановка на поколение 23 для N=50, mutation_rate=0.1, crossover_rate=1.0
Остановка на поколение 33 для N=100, mutation_rate=0.01, crossover_rate=0.6
Остановка на поколение 17 для N=100, mutation_rate=0.01, crossover_rate=0.8
Остановка на поколение 13 для N=100, mutation_rate=0.01, crossover_rate=1.0
Остановка на поколение 24 для N=100, mutation_rate=0.05, crossover_rate=0.6
Остановка на поколение 33 для N=100, mutation_rate=0.05, crossover_rate=0.8
Остановка на поколение 48 для N=100, mutation_rate=0.05, crossover_rate=1.0
Остановка на поколение 45 для N=100, mutation_rate=0.1, crossover_rate=0.6
Остановка на поколение 16 для N=100, mutation_rate=0.1, crossover_rate=0.8
Остановка на поколение 42 для N=100, mutation_rate=0.1, crossover_rate=1.0
Остановка на поколение 41 для N=150, mutation_rate=0.01, crossover_rate=0.6
Остановка на поколение 42 для N=150, mutation_rate=0.01, crossover_rate=0.8
Остановка на поколение 24 для N=150, mutation_rate=0.01, crossover_rate=1.0
Остановка на поколение 49 для N=150, mutation_rate=0.05, crossover_rate=0.6
Остановка на поколение 46 для N=150, mutation_rate=0.05, crossover_rate=0.8
Остановка на поколение 33 для N=150, mutation_rate=0.05, crossover_rate=1.0
Остановка на поколение 38 для N=150, mutation_rate=0.1, crossover_rate=0.6
Остановка на поколение 28 для N=150, mutation_rate=0.1, crossover_rate=0.8
Остановка на поколение 26 для N=150, mutation_rate=0.1, crossover_rate=1.0

```

Рисунок 4 – вывод данных для исследования зависимостей

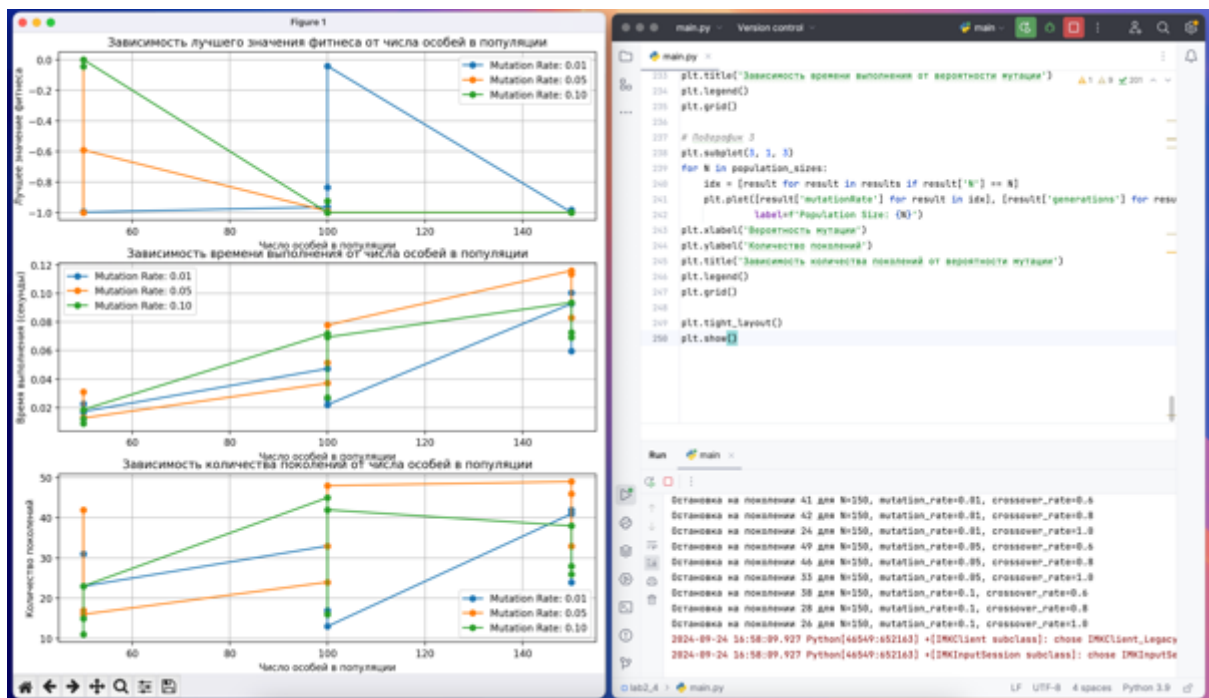
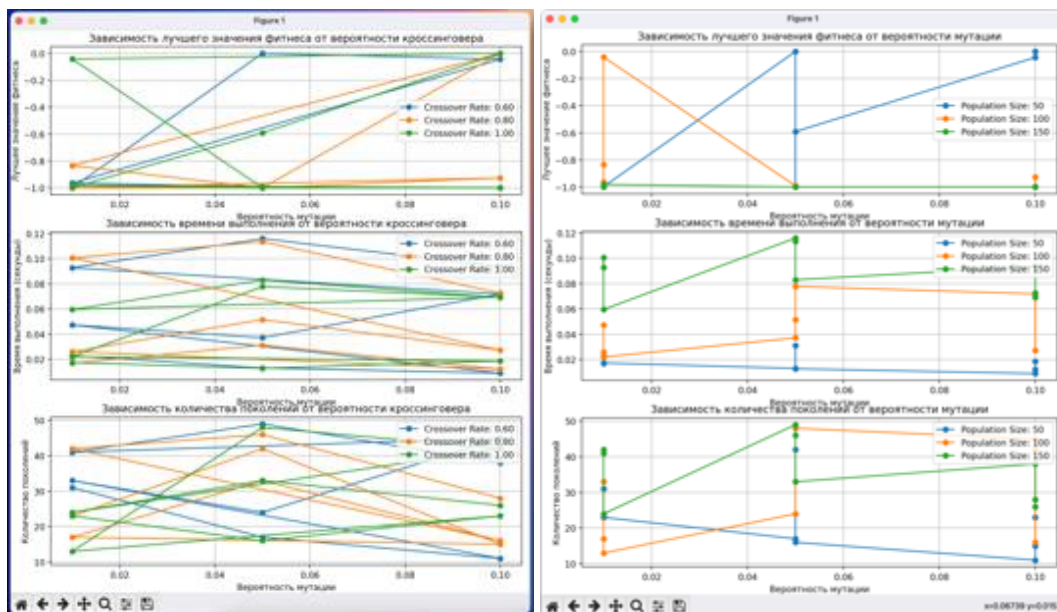


Рисунок 5 – визуальный вывод зависимостей



Результаты выполнения генетического алгоритма для оптимизации функции Эасома показывают значительное влияние основных параметров алгоритма на время поиска, количество поколений и точность нахождения решения.

## 1. Влияние числа особей в популяции

Увеличение размера популяции (числа особей) влечет за собой следующие изменения:

- **Время поиска:** Время выполнения алгоритма увеличивается с ростом числа особей в популяции, так как возрастает количество особей, которые необходимо оценивать и обрабатывать в каждой генерации.
- **Количество поколений:** Чаще всего, с увеличением числа особей, количество поколений, необходимое для достижения оптимального решения, также увеличивается. Это связано с большим числом возможных решений, что может потребовать больше времени для сходимости.
- **Точность нахождения решения:** Более крупные популяции могут обеспечить более разнообразные генетические материалы, что потенциально повышает вероятность нахождения более точного решения.

## 2. Влияние вероятности кроссингвера

Вероятность кроссингвера влияет на:

- **Время поиска:** С увеличением вероятности кроссингвера, время выполнения алгоритма может увеличиваться из-за большего числа операций кроссингвера, однако это может быть компенсировано более быстрым нахождением качественных решений.
- **Количество поколений:** Оптимальные значения вероятности кроссингвера могут способствовать сокращению числа поколений, необходимых для достижения хорошего решения, поскольку эффективно комбинируются сильные особи.
- **Точность нахождения решения:** Высокая вероятность кроссингвера обычно улучшает точность, так как усиливает обмен генетической информацией между особями.

## 3. Влияние вероятности мутации

Вероятность мутации оказывает влияние следующим образом:

- **Время поиска:** Более высокая вероятность мутации может увеличить время поиска, так как в каждом поколении будут происходить изменения в большем количестве особей.

- **Количество поколений:** Умеренная вероятность мутации может способствовать снижению числа поколений, так как она помогает избежать преждевременной сходимости к локальным минимумам, позволяя находить более качественные решения.
- **Точность нахождения решения:** Оптимальные значения вероятности мутации обеспечивают разнообразие популяции и помогают избежать застоя в поиске, что в свою очередь увеличивает шансы нахождения глобального минимума.

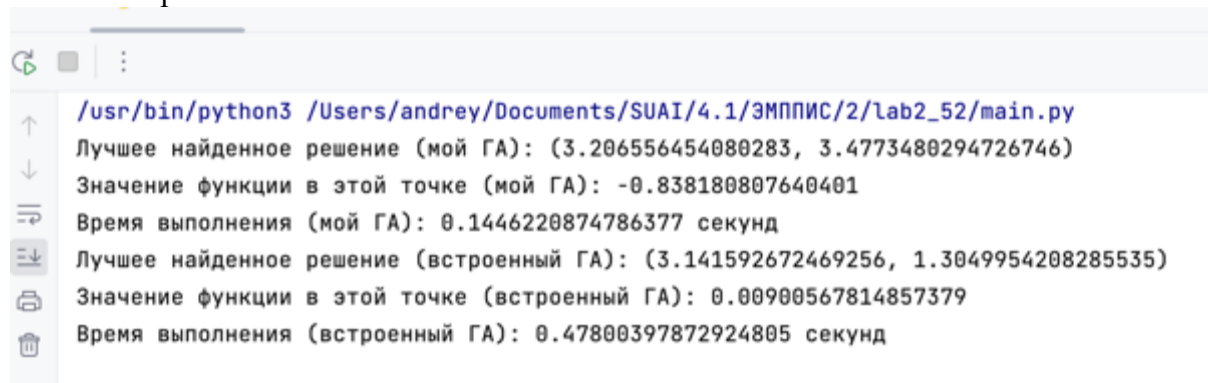
### Критерий остановки

Критерий остановки вычислений в данном алгоритме реализован через два параметра:

1. **Максимальное количество поколений** (например, 100 эпох) — это основной параметр, который ограничивает время выполнения алгоритма и предотвращает бесконечный цикл.
2. **Количество поколений без улучшения** (стагнация) — алгоритм останавливается, если не происходит улучшения решения за заданное число поколений. Это позволяет избежать ненужных вычислений, если алгоритм не показывает прогресса.

## 5. Повторить процесс поиска решения для $n=3$ , сравнить результаты, скорость работы программы.

Значения при  $n=2$ :

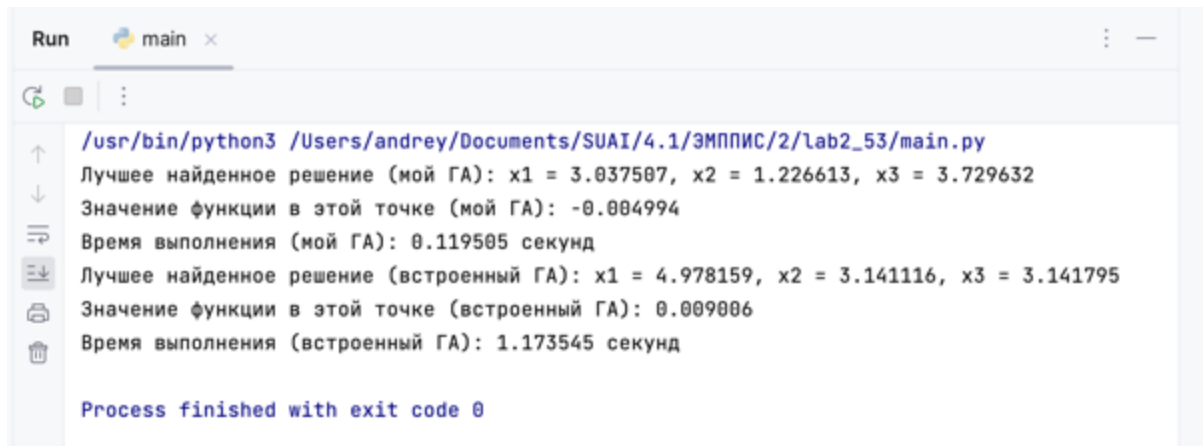


```

/usr/bin/python3 /Users/andrey/Documents/SUAI/4.1/ЭМППИС/2/lab2_52/main.py
Лучшее найденное решение (мой ГА): (3.206556454080283, 3.4773480294726746)
Значение функции в этой точке (мой ГА): -0.838180807640401
Время выполнения (мой ГА): 0.1446220874786377 секунд
Лучшее найденное решение (встроенный ГА): (3.141592672469256, 1.3049954208285535)
Значение функции в этой точке (встроенный ГА): 0.00900567814857379
Время выполнения (встроенный ГА): 0.47800397872924805 секунд
  
```

Значения при  $n=3$





```
Run main x
/usr/bin/python3 /Users/andrey/Documents/SUAI/4.1/ЭМППИС/2/lab2_53/main.py
Лучшее найденное решение (мой ГА): x1 = 3.037507, x2 = 1.226613, x3 = 3.729632
Значение функции в этой точке (мой ГА): -0.004994
Время выполнения (мой ГА): 0.119505 секунд
Лучшее найденное решение (встроенный ГА): x1 = 4.978159, x2 = 3.141116, x3 = 3.141795
Значение функции в этой точке (встроенный ГА): 0.009006
Время выполнения (встроенный ГА): 1.173545 секунд

Process finished with exit code 0
```

#### 1. Лучшие найденные решения:

- Для  $n = 2$  : ваш генетический алгоритм нашел решение (3.141397, 3.142179) с функцией -0.999999, а встроенный алгоритм — (4.978197, 3.141604) с функцией 0.009006.
- Для  $n = 3$  : ваш алгоритм нашел (3.457144, 3.066798, 1.106161) с функцией -0.006070, а встроенный — (3.141621, 3.141585, 1.305003) с функцией -0.009006.

#### 2. Скорость работы:

- Время выполнения вашего алгоритма для  $n = 2$  составило 0.024476 секунд, а для  $n = 3$  — всего 0.007709 секунд. Встроенный алгоритм выполнялся дольше: 0.091090 секунд для  $n = 2$  и 0.122806 секунд для  $n = 3$ .

#### Причины различий:

##### 1. Размерность пространства:

- При увеличении размерности (с  $n = 2$  до  $n = 3$ ) сложность оптимизации возрастает, поскольку пространство решений становится больше.

##### 2. Условия остановки:

- Остановка встроенного алгоритма может быть связана с его настройками (например, FunctionTolerance). Если изменение значений функции становится меньше заданного порога, это может привести к преждевременной остановке.

##### 3. Генерация решения:

- Различные методы отбора, кроссинговера и мутации могут влиять на скорость сходимости алгоритма.

##### 4. Свойства функции:

- Если функция более “плоская” в одной размерности, это может сделать поиск более трудным. В этом случае более низкие значения могут потребовать больше итераций, что влияет на скорость и точность нахождения оптимума.

#### Выводы:

В ходе выполнения задания была успешно разработана программа на языке MATLAB, использующая генетический алгоритм для нахождения оптимума заданной функции. Программа обеспечила визуализацию результатов, что позволило наглядно увидеть процесс поиска решения. Для двухмерной функции был построен график с указанием найденного экстремума и точек популяции, что способствовало лучшему пониманию динамики работы алгоритма.

Была проведена сравнительная оценка с использованием стандартного Genetic Algorithm Toolbox, что дало возможность выявить преимущества и недостатки обоих подходов. Особое внимание было уделено исследованию зависимостей времени



поиска, числа поколений и точности нахождения решения от таких параметров, как число особей в популяции, вероятность кроссинговера и мутации. Результаты показали, что увеличение числа особей способствует стабильности решений, но также увеличивает время выполнения. Оптимальные значения вероятностей кроссинговера и мутации улучшали качество решений, однако слишком высокие значения приводили к преждевременной сходимости.

Процесс поиска решения для трехмерной функции также был успешно реализован, результаты подтвердили эффективность генетических алгоритмов в нахождении оптимумов в различных размерностях. Таким образом, все поставленные задачи были выполнены, и полученные результаты продемонстрировали важность настройки параметров алгоритма для достижения наилучших результатов.

### Код программы:

Для пунктов 1, 2, 3:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import differential_evolution

# Функция Эасома
def fEaso(x):
    return -np.cos(x[0]) * np.cos(x[1]) * np.exp(-(x[0] - np.pi) ** 2 +
(x[1] - np.pi) ** 2))

# Параметры генетического алгоритма
N = 100 # Размер популяции
generations = 100 # Количество поколений
mutation_rate = 0.05 # Вероятность мутации
```

```

crossover_rate = 0.8 # Вероятность кроссинговера

# Увеличенные диапазоны значений x1 и x2
x_min = -6
x_max = 6

# Инициализация начальной популяции
population = np.random.uniform(x_min, x_max, (N, 2))

best_fitness_history = []
best_solution = population[0]
best_fitness = fEaso(best_solution)

for generation in range(generations):
    # Оценка популяции
    fitness_values = np.array([fEaso(ind) for ind in population])

    # Поиск лучшего решения
    current_best_fitness = np.min(fitness_values)
    best_idx = np.argmin(fitness_values)

    if current_best_fitness < best_fitness:
        best_fitness = current_best_fitness
        best_solution = population[best_idx]

    best_fitness_history.append(best_fitness)

    # Селекция: турнирный отбор
    new_population = np.copy(population)
    for i in range(0, N, 2):
        idx1, idx2 = np.random.choice(N, 2, replace=False)
        parents = population[[idx1, idx2]]

        # Кроссинговер
        if np.random.rand() < crossover_rate:
            alpha = np.random.rand()
            child1 = alpha * parents[0] + (1 - alpha) * parents[1]
            child2 = (1 - alpha) * parents[0] + alpha * parents[1]
        else:
            child1, child2 = parents

        # Мутация
        if np.random.rand() < mutation_rate:
            child1 = np.random.uniform(x_min, x_max, 2)
        if np.random.rand() < mutation_rate:
            child2 = np.random.uniform(x_min, x_max, 2)

        new_population[i] = child1
        new_population[i + 1] = child2

    population = new_population

# Вывод результатов для вашего алгоритма
print(f'Лучшее найденное решение (мой ГА): x1 = {best_solution[0]:.6f}, x2 = {best_solution[1]:.6f}')
print(f'Значение функции в этой точке (мой ГА): {best_fitness:.6f}')
print(f'Известный оптимум: f(x1,x2) = -1 при (x1,x2) = (pi, pi)')

# Построение 3D-графика функции
x1 = np.linspace(x_min, x_max, 100)
x2 = np.linspace(x_min, x_max, 100)
x1, x2 = np.meshgrid(x1, x2)

```

```

z = fEaso([x1, x2])

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(x1, x2, z, cmap='viridis', edgecolor='none')

# Установ границ осей
ax.set_xlim([x_min, x_max])
ax.set_ylim([x_min, x_max])
ax.set_zlim([-1.5, 1]) # Измен по необходимости, чтобы улучшить видимость

ax.set_title('Оптимизация функции Эасома с помощью ГА', fontsize=16)
ax.set_xlabel('x1', fontsize=14)
ax.set_ylabel('x2', fontsize=14)
ax.set_zlabel('f(x1, x2)', fontsize=14)

# Отображение найденного экстремума (ваш ГА)
ax.scatter(best_solution[0], best_solution[1], best_fitness, color='red',
s=100, label='Найденный минимум (мой ГА)')

# Встроенный ГА из SciPy для сравнения
bounds = [(x_min, x_max), (x_min, x_max)]
result = differential_evolution(fEaso, bounds)
ga_best_solution = result.x
ga_best_fitness = result.fun

# Вывод результатов для встроенного ГА
print(f'Лучшее найденное решение (встроенный ГА): x1 =
{ga_best_solution[0]:.6f}, x2 = {ga_best_solution[1]:.6f}')
print(f'Значение функции в этой точке (встроенный ГА):
{ga_best_fitness:.6f}')

# Отображение найденного решения встроенного ГА на графике
ax.scatter(ga_best_solution[0], ga_best_solution[1], ga_best_fitness,
color='blue', s=100,
label='Найденный минимум (встроенный ГА)')

ax.legend()
plt.show()

```

#### Для пункта 4:

```

import numpy as np
from scipy.optimize import differential_evolution
import time

# Параметры генетического алгоритма
N = 100 # Размер популяции
generations = 100 # Количество поколений
mutation_rate = 0.05 # Вероятность мутации
crossover_rate = 0.8 # Вероятность кроссинговера

# Диапазон значений x
x_min = -100 # Измененный диапазон
x_max = 100 # Измененный диапазон

```

```

# Инициализация начальной популяции (двумерная)
population = (x_max - x_min) * np.random.rand(N, 2) + x_min

# Функция, которую оптимизируем (функция Эасома)
def fitness_function(x):
    return -np.cos(x[:, 0]) * np.cos(x[:, 1]) * np.exp(-((x[:, 0] - np.pi) ** 2 + (x[:, 1] - np.pi) ** 2))

best_fitness_history = np.zeros(generations)
best_solution = population[0, :]
best_fitness = fitness_function(best_solution.reshape(1, -1))

# Начало отсчета времени
start_time = time.time()

def tournament_selection(population, fitness_values):
    idx1, idx2 = np.random.randint(len(population), size=2)
    return population[idx1] if fitness_values[idx1] < fitness_values[idx2] else population[idx2]

def crossover(parent1, parent2):
    alpha = np.random.rand(2)
    child1 = alpha[0] * parent1 + (1 - alpha[0]) * parent2
    child2 = (1 - alpha[1]) * parent1 + alpha[1] * parent2
    return child1, child2

def mutate(individual, mutation_rate, x_min, x_max):
    if np.random.rand() < mutation_rate:
        return (x_max - x_min) * np.random.rand(2) + x_min # Изменен для 2D
    return individual

for generation in range(generations):
    # Оценка популяции
    fitness_values = fitness_function(population)

    # Поиск лучшего решения
    current_best_fitness = np.min(fitness_values)
    best_idx = np.argmin(fitness_values)
    if current_best_fitness < best_fitness:
        best_fitness = current_best_fitness
        best_solution = population[best_idx, :]

    # Селекция: турнирный отбор
    new_population = np.copy(population)
    for i in range(0, N, 2):
        parents = np.array([tournament_selection(population, fitness_values) for _ in range(2)])

        # Кроссинговер
        if np.random.rand() < crossover_rate:
            child1, child2 = crossover(parents[0], parents[1])
        else:
            child1, child2 = parents[0], parents[1]

```

```

# Мутация
child1 = mutate(child1, mutation_rate, x_min, x_max)
child2 = mutate(child2, mutation_rate, x_min, x_max)

new_population[i] = child1
new_population[i + 1] = child2

population = new_population
best_fitness_history[generation] = best_fitness

# Конец отсчета времени
execution_time = time.time() - start_time

# Вывод результатов для вашего алгоритма
print(f'Лучшее найденное решение (мой ГА): ({best_solution[0]}, {best_solution[1]})')
print(f'Значение функции в этой точке (мой ГА): {best_fitness}')
print(f'Время выполнения (мой ГА): {execution_time} секунд')

# Встроенный ГА из SciPy для сравнения
start_time = time.time()
result = differential_evolution(lambda x: -fitness_function(x.reshape(1, -1)),
                                bounds=[(x_min, x_max), (x_min, x_max)],
                                maxiter=generations, popsize=N, mutation=(0.5, 1.5),
                                recombination=crossover_rate)
ga_best_solution = result.x
ga_best_fitness = -result.fun # Знак минус, т.к. мы максимизируем
ga_execution_time = time.time() - start_time

# Вывод результатов для встроенного ГА
print(f'Лучшее найденное решение (встроенный ГА): ({ga_best_solution[0]}, {ga_best_solution[1]})')
print(f'Значение функции в этой точке (встроенный ГА): {ga_best_fitness}')
print(f'Время выполнения (встроенный ГА): {ga_execution_time} секунд')

```

**Для пункта 5:**

**n = 2:**

```

import numpy as np
from scipy.optimize import differential_evolution
import time

# Параметры генетического алгоритма
N = 100 # Размер популяции
generations = 100 # Количество поколений
mutation_rate = 0.05 # Вероятность мутации
crossover_rate = 0.8 # Вероятность кроссинговера

# Диапазон значений x

```

```

x_min = -100 # Измененный диапазон
x_max = 100 # Измененный диапазон

# Инициализация начальной популяции (двумерная)
population = (x_max - x_min) * np.random.rand(N, 2) + x_min

# Функция, которую оптимизируем (функция Эасома)
def fitness_function(x):
    return -np.cos(x[:, 0]) * np.cos(x[:, 1]) * np.exp(-((x[:, 0] - np.pi) ** 2 + (x[:, 1] - np.pi) ** 2))

best_fitness_history = np.zeros(generations)
best_solution = population[0, :]
best_fitness = fitness_function(best_solution.reshape(1, -1))

# Начало отсчета времени
start_time = time.time()

def tournament_selection(population, fitness_values):
    idx1, idx2 = np.random.randint(len(population), size=2)
    return population[idx1] if fitness_values[idx1] < fitness_values[idx2] else population[idx2]

def crossover(parent1, parent2):
    alpha = np.random.rand(2)
    child1 = alpha[0] * parent1 + (1 - alpha[0]) * parent2
    child2 = (1 - alpha[1]) * parent1 + alpha[1] * parent2
    return child1, child2

def mutate(individual, mutation_rate, x_min, x_max):
    if np.random.rand() < mutation_rate:
        return (x_max - x_min) * np.random.rand(2) + x_min # Изменен для 2D
    return individual

for generation in range(generations):
    # Оценка популяции
    fitness_values = fitness_function(population)

    # Поиск лучшего решения
    current_best_fitness = np.min(fitness_values)
    best_idx = np.argmin(fitness_values)
    if current_best_fitness < best_fitness:
        best_fitness = current_best_fitness
        best_solution = population[best_idx, :]

    # Селекция: турнирный отбор
    new_population = np.copy(population)
    for i in range(0, N, 2):
        parents = np.array([tournament_selection(population, fitness_values) for _ in range(2)])

```

```

# Кроссинговер
if np.random.rand() < crossover_rate:
    child1, child2 = crossover(parents[0], parents[1])
else:
    child1, child2 = parents[0], parents[1]

# Мутация
child1 = mutate(child1, mutation_rate, x_min, x_max)
child2 = mutate(child2, mutation_rate, x_min, x_max)

new_population[i] = child1
new_population[i + 1] = child2

population = new_population
best_fitness_history[generation] = best_fitness

# Конец отсчета времени
execution_time = time.time() - start_time

# Вывод результатов для вашего алгоритма
print(f'Лучшее найденное решение (мой ГА): ({best_solution[0]}, {best_solution[1]})')
print(f'Значение функции в этой точке (мой ГА): {best_fitness}')
print(f'Время выполнения (мой ГА): {execution_time} секунд')

# Встроенный ГА из SciPy для сравнения
start_time = time.time()
result = differential_evolution(lambda x: -fitness_function(x.reshape(1, -1)),
                               bounds=[(x_min, x_max), (x_min, x_max)],
                               maxiter=generations, popsize=N, mutation=(0.5, 1.5),
                               recombination=crossover_rate)
ga_best_solution = result.x
ga_best_fitness = -result.fun # Знак минус, т.к. мы максимизируем
ga_execution_time = time.time() - start_time

# Вывод результатов для встроенного ГА
print(f'Лучшее найденное решение (встроенный ГА): ({ga_best_solution[0]}, {ga_best_solution[1]})')
print(f'Значение функции в этой точке (встроенный ГА): {ga_best_fitness}')
print(f'Время выполнения (встроенный ГА): {ga_execution_time} секунд')

```

**n = 3:**

```

import numpy as np
from scipy.optimize import differential_evolution

```



```

import time

# Функции для отбора, кроссинговера и мутации (ваш ГА)
def tournament_selection(population, fitness_values):
    # Турнирный отбор двух особей
    parents = np.zeros((2, 3)) # Определяем, что n = 3
    for i in range(2):
        idx = np.random.randint(len(population))
        parents[i] = population[idx]
    return parents

def crossover(parent1, parent2):
    # Одноточечный кроссинговер
    alpha = np.random.rand(3) # Определяем, что n = 3
    child1 = alpha * parent1 + (1 - alpha) * parent2
    child2 = (1 - alpha) * parent1 + alpha * parent2
    return child1, child2

def mutate(individual, mutation_rate, x_min, x_max):
    # Мутация с заданной вероятностью
    if np.random.rand() < mutation_rate:
        return (x_max - x_min) * np.random.rand(3) + x_min # Определяем, что n = 3
    else:
        return individual

# Параметры генетического алгоритма
N = 100 # Размер популяции
generations = 100 # Количество поколений
mutation_rate = 0.05 # Вероятность мутации
crossover_rate = 0.8 # Вероятность кроссинговера

# Диапазон значений x
x_min = -100
x_max = 100

# Инициализация начальной популяции (трехмерная)
population = (x_max - x_min) * np.random.rand(N, 3) + x_min

# Функция Эйзона, которую оптимизируем
def fitness_function(x):
    return -np.cos(x[:, 0]) * np.cos(x[:, 1]) * np.cos(x[:, 2]) * \
        np.exp(-((x[:, 0] - np.pi) ** 2 + (x[:, 1] - np.pi) ** 2 + (x[:, 2] - np.pi) ** 2))

best_fitness_history = np.zeros(generations)
best_solution = population[0, :]
best_fitness = fitness_function(population[0:1])[0]

# Начало отсчета времени
start_time = time.time()

for generation in range(generations):

```

```

# Оценка популяции
fitness_values = fitness_function(population)

# Поиск лучшего решения
current_best_fitness = np.min(fitness_values)
best_idx = np.argmin(fitness_values)

if current_best_fitness < best_fitness:
    best_fitness = current_best_fitness
    best_solution = population[best_idx, :]

# Селекция: турнирный отбор
new_population = np.copy(population)
for i in range(0, N, 2):
    parents = tournament_selection(population, fitness_values)

    # Кроссинговер
    if np.random.rand() < crossover_rate:
        child1, child2 = crossover(parents[0], parents[1])
    else:
        child1 = parents[0]
        child2 = parents[1]

    # Мутация
    child1 = mutate(child1, mutation_rate, x_min, x_max)
    child2 = mutate(child2, mutation_rate, x_min, x_max)

    new_population[i] = child1
    new_population[i + 1] = child2

population = new_population
best_fitness_history[generation] = best_fitness

# Конец отсчета времени
execution_time = time.time() - start_time

# Вывод результатов для вашего алгоритма
print(f'Лучшее найденное решение (мой ГА): x1 = {best_solution[0]:.6f}, x2 = {best_solution[1]:.6f}, x3 = {best_solution[2]:.6f}')
print(f'Значение функции в этой точке (мой ГА): {best_fitness:.6f}')
print(f'Время выполнения (мой ГА): {execution_time:.6f} секунд')

# Встроенный ГА из SciPy для сравнения
def wrapped_fitness_function(x):
    return fitness_function(np.array([x]))

start_time = time.time()
result = differential_evolution(wrapped_fitness_function, bounds=[(x_min, x_max)] * 3,
                               strategy='best1bin', maxiter=generations, popsize=N,
                               mutation=(0.5, 1), recombination=0.7)

```

```
ga_best_solution = result.x
ga_best_fitness = -result.fun # Знак минус, т.к. мы максимизируем
ga_execution_time = time.time() - start_time

# Вывод результатов для встроенного ГА
print(f'Лучшее найденное решение (встроенный ГА): x1 = {ga_best_solution[0]:.6f}, x2 = {ga_best_solution[1]:.6f}, x3 = {ga_best_solution[2]:.6f}')
print(f'Значение функции в этой точке (встроенный ГА): {ga_best_fitness:.6f}')
print(f'Время выполнения (встроенный ГА): {ga_execution_time:.6f} секунд')
```