ГУАП

КАФЕДРА № 43

ОТЧЕТ  
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| профессор |  |  |  | Ю.А. Скобцов |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №2 |
| Простой генетический алгоритм |
| по курсу: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

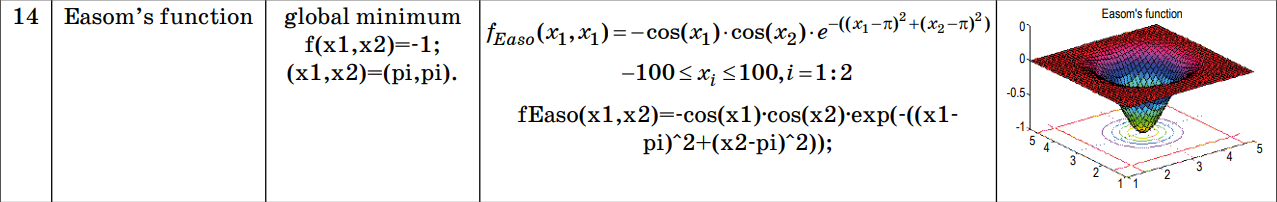
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4134 |  |  |  | Д. В. Самарин |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2024

**Цель:**

Модификация представления хромосомы и операторов рекомбинации ГА для оптимизации многомерных функций. Графическое отображение результатов оптимизации.

**Вариант 14.**



# 1. 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.

# Программа

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 Эйзома, 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')

#Исследование

import numpy as np

import matplotlib.pyplot as plt

import time

# Параметры генетического алгоритма

population\_sizes = [50, 100, 150]  # Размеры популяции для тестирования

mutation\_rates = [0.01, 0.05, 0.1]  # Вероятности мутации для тестирования

crossover\_rates = [0.6, 0.8, 1.0]  # Вероятности кроссинговера для тестирования

max\_stagnation = 10  # Максимальное количество поколений без улучшения

max\_generations = 100  # Максимальное количество поколений

# Диапазон значений x

x\_min = -100

x\_max = 100

# 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))

# Функция турнирового отбора

def tournament\_selection(population, fitness\_values, tournament\_size=3):

    selected\_parents = []

    for \_ in range(2):  # Селектируем двух родителей

        tournament\_indices = np.random.choice(len(population), tournament\_size, replace=False)

        tournament\_fitness = fitness\_values[tournament\_indices]

        winner\_index = tournament\_indices[np.argmin(tournament\_fitness)]  # Минимизация

        selected\_parents.append(population[winner\_index])

    return np.array(selected\_parents)

# Функция кроссинговера

def crossover(parent1, parent2):

    alpha = np.random.rand()  # Случайный коэффициент для интерполяции

    child1 = alpha \* parent1 + (1 - alpha) \* parent2

    child2 = (1 - alpha) \* parent1 + alpha \* parent2

    return child1, child2

# Функция мутации

def mutate(individual, mutation\_rate, x\_min, x\_max):

    if np.random.rand() < mutation\_rate:

        mutation\_value = np.random.uniform(x\_min, x\_max, size=individual.shape)

        individual += mutation\_value  # Применение мутации

        individual = np.clip(individual, x\_min, x\_max)  # Ограничение в пределах допустимого диапазона

    return individual

# Для хранения результатов

results = []

# Проход по всем параметрам

for N in population\_sizes:

    for mutation\_rate in mutation\_rates:

        for crossover\_rate in crossover\_rates:

            # Инициализация начальной популяции

            population = np.random.uniform(x\_min, x\_max, (N, 2))

            best\_fitness\_history = np.zeros(max\_generations)

            best\_solution = population[0, :]

            best\_fitness = fitness\_function(population[0, :].reshape(1, -1))

            stagnation\_counter = 0

            # Начало измерения времени

            start\_time = time.time()

            for generation in range(max\_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, :]

                    stagnation\_counter = 0  # Сброс счетчика застоя

                else:

                    stagnation\_counter += 1  # Увеличение счетчика застоя

                # Селекция: турнирный отбор

                new\_population = population.copy()

                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

                # Проверка условий остановки

                if stagnation\_counter >= max\_stagnation:

                    print(

                        f'Остановка на поколении {generation} для N={N}, mutation\_rate={mutation\_rate}, crossover\_rate={crossover\_rate}')

                    break

            # Конец измерения времени

            elapsed\_time = time.time() - start\_time

            # Сохранение результатов

            results.append({

                'N': N,

                'mutationRate': mutation\_rate,

                'crossoverRate': crossover\_rate,

                'bestSolution': best\_solution,

                'bestFitness': best\_fitness,

                'elapsedTime': elapsed\_time,

                'generations': generation

            })

# Визуализация результатов для числа особей в популяции

plt.figure(figsize=(10, 12))

# Подграфик 1

plt.subplot(3, 1, 1)

for mutation\_rate in mutation\_rates:

    idx = [result for result in results if result['mutationRate'] == mutation\_rate]

    plt.plot([result['N'] for result in idx], [result['bestFitness'] for result in idx], '-o',

             label=f'Mutation Rate: {mutation\_rate:.2f}')

plt.xlabel('Число особей в популяции')

plt.ylabel('Лучшее значение фитнеса')

plt.title('Зависимость лучшего значения фитнеса от числа особей в популяции')

plt.legend()

plt.grid()

# Подграфик 2

plt.subplot(3, 1, 2)

for mutation\_rate in mutation\_rates:

    idx = [result for result in results if result['mutationRate'] == mutation\_rate]

    plt.plot([result['N'] for result in idx], [result['elapsedTime'] for result in idx], '-o',

             label=f'Mutation Rate: {mutation\_rate:.2f}')

plt.xlabel('Число особей в популяции')

plt.ylabel('Время выполнения (секунды)')

plt.title('Зависимость времени выполнения от числа особей в популяции')

plt.legend()

plt.grid()

# Подграфик 3

plt.subplot(3, 1, 3)

for mutation\_rate in mutation\_rates:

    idx = [result for result in results if result['mutationRate'] == mutation\_rate]

    plt.plot([result['N'] for result in idx], [result['generations'] for result in idx], '-o',

             label=f'Mutation Rate: {mutation\_rate:.2f}')

plt.xlabel('Число особей в популяции')

plt.ylabel('Количество поколений')

plt.title('Зависимость количества поколений от числа особей в популяции')

plt.legend()

plt.grid()

plt.tight\_layout()

plt.show()

# Визуализация результатов для вероятности кроссинговера

plt.figure(figsize=(10, 12))

# Подграфик 1

plt.subplot(3, 1, 1)

for crossover\_rate in crossover\_rates:

    idx = [result for result in results if result['crossoverRate'] == crossover\_rate]

    plt.plot([result['mutationRate'] for result in idx], [result['bestFitness'] for result in idx], '-o',

             label=f'Crossover Rate: {crossover\_rate:.2f}')

plt.xlabel('Вероятность мутации')

plt.ylabel('Лучшее значение фитнеса')

plt.title('Зависимость лучшего значения фитнеса от вероятности кроссинговера')

plt.legend()

plt.grid()

# Подграфик 2

plt.subplot(3, 1, 2)

for crossover\_rate in crossover\_rates:

    idx = [result for result in results if result['crossoverRate'] == crossover\_rate]

    plt.plot([result['mutationRate'] for result in idx], [result['elapsedTime'] for result in idx], '-o',

             label=f'Crossover Rate: {crossover\_rate:.2f}')

plt.xlabel('Вероятность мутации')

plt.ylabel('Время выполнения (секунды)')

plt.title('Зависимость времени выполнения от вероятности кроссинговера')

plt.legend()

plt.grid()

# Подграфик 3

plt.subplot(3, 1, 3)

for crossover\_rate in crossover\_rates:

    idx = [result for result in results if result['crossoverRate'] == crossover\_rate]

    plt.plot([result['mutationRate'] for result in idx], [result['generations'] for result in idx], '-o',

             label=f'Crossover Rate: {crossover\_rate:.2f}')

plt.xlabel('Вероятность мутации')

plt.ylabel('Количество поколений')

plt.title('Зависимость количества поколений от вероятности кроссинговера')

plt.legend()

plt.grid()

plt.tight\_layout()

plt.show()

# Визуализация результатов для вероятности мутации

plt.figure(figsize=(10, 12))

# Подграфик 1

plt.subplot(3, 1, 1)

for N in population\_sizes:

    idx = [result for result in results if result['N'] == N]

    plt.plot([result['mutationRate'] for result in idx], [result['bestFitness'] for result in idx], '-o',

             label=f'Population Size: {N}')

plt.xlabel('Вероятность мутации')

plt.ylabel('Лучшее значение фитнеса')

plt.title('Зависимость лучшего значения фитнеса от вероятности мутации')

plt.legend()

plt.grid()

# Подграфик 2

plt.subplot(3, 1, 2)

for N in population\_sizes:

    idx = [result for result in results if result['N'] == N]

    plt.plot([result['mutationRate'] for result in idx], [result['elapsedTime'] for result in idx], '-o',

             label=f'Population Size: {N}')

plt.xlabel('Вероятность мутации')

plt.ylabel('Время выполнения (секунды)')

plt.title('Зависимость времени выполнения от вероятности мутации')

plt.legend()

plt.grid()

# Подграфик 3

plt.subplot(3, 1, 3)

for N in population\_sizes:

    idx = [result for result in results if result['N'] == N]

    plt.plot([result['mutationRate'] for result in idx], [result['generations'] for result in idx], '-o',

             label=f'Population Size: {N}')

plt.xlabel('Вероятность мутации')

plt.ylabel('Количество поколений')

plt.title('Зависимость количества поколений от вероятности мутации')

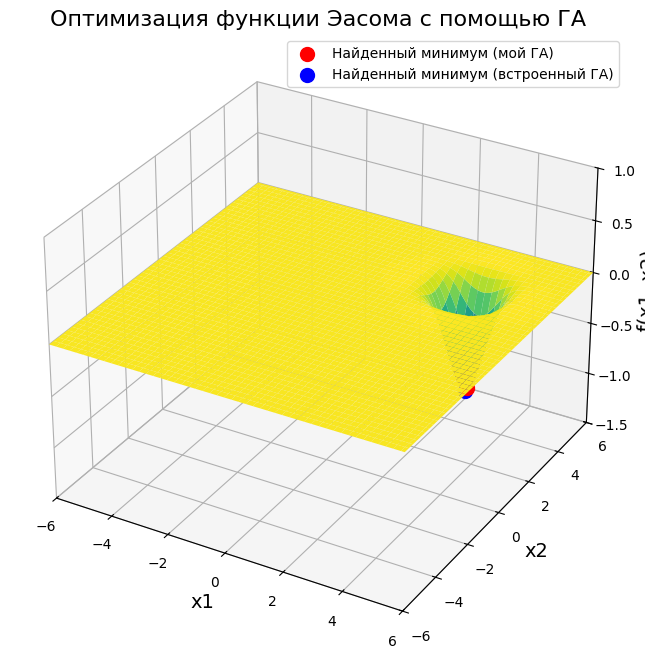
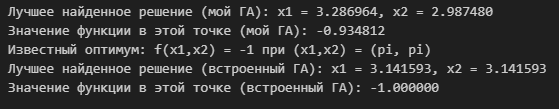
plt.legend()

plt.grid()

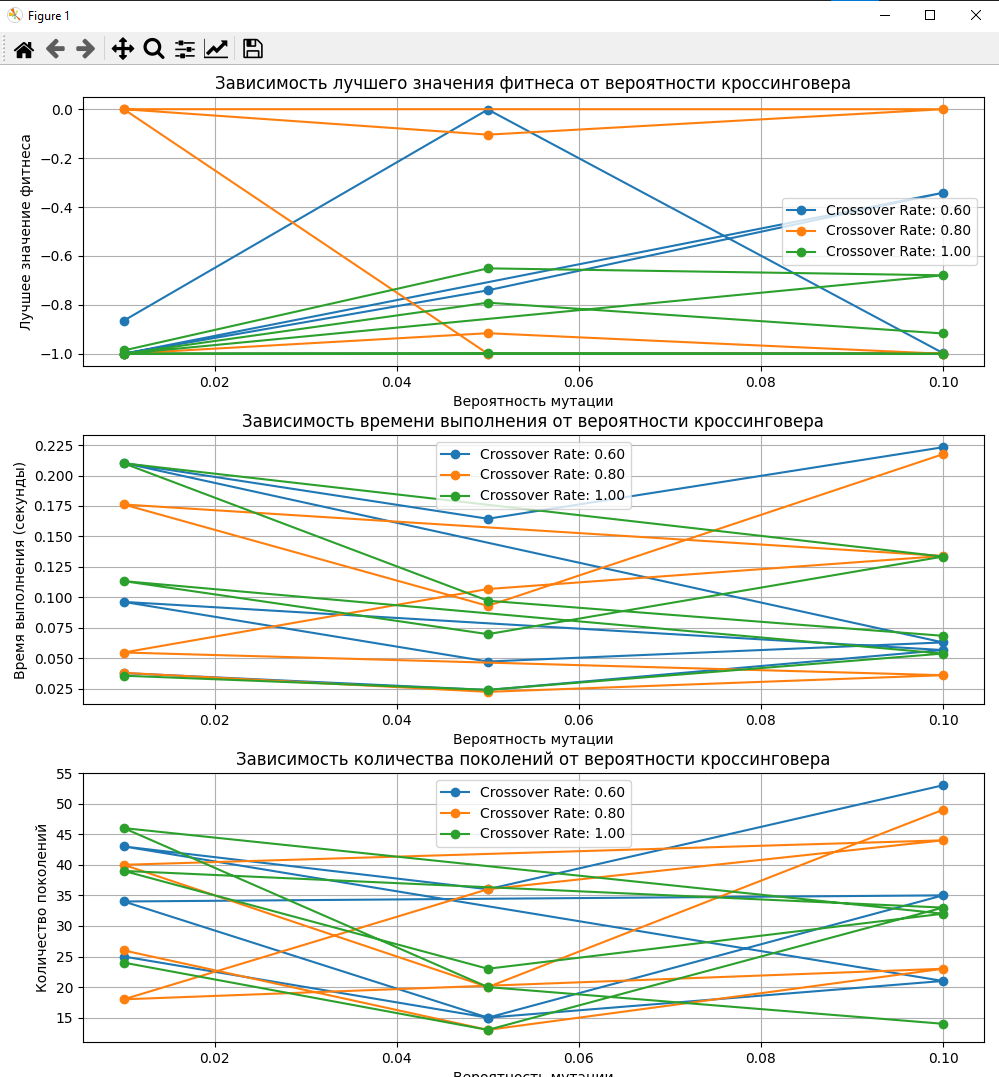
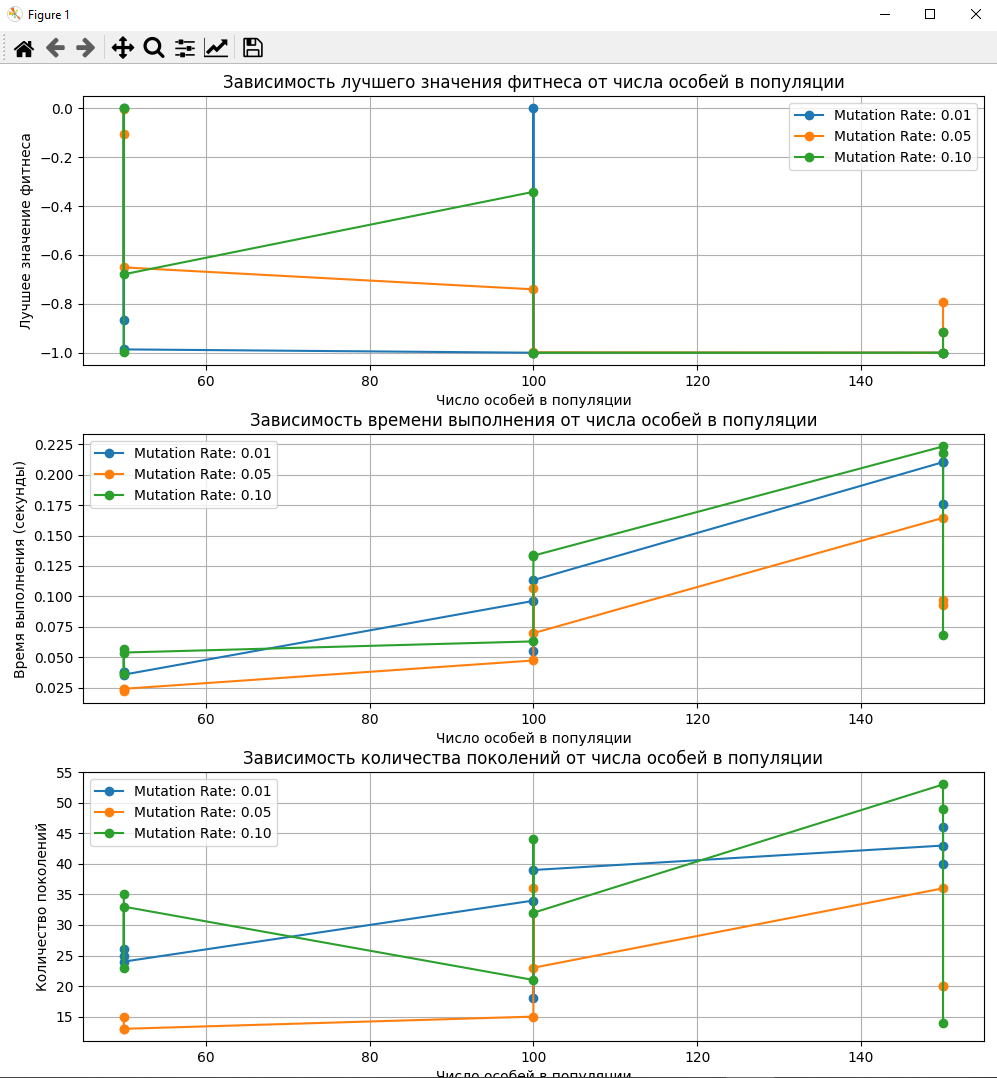
plt.tight\_layout()

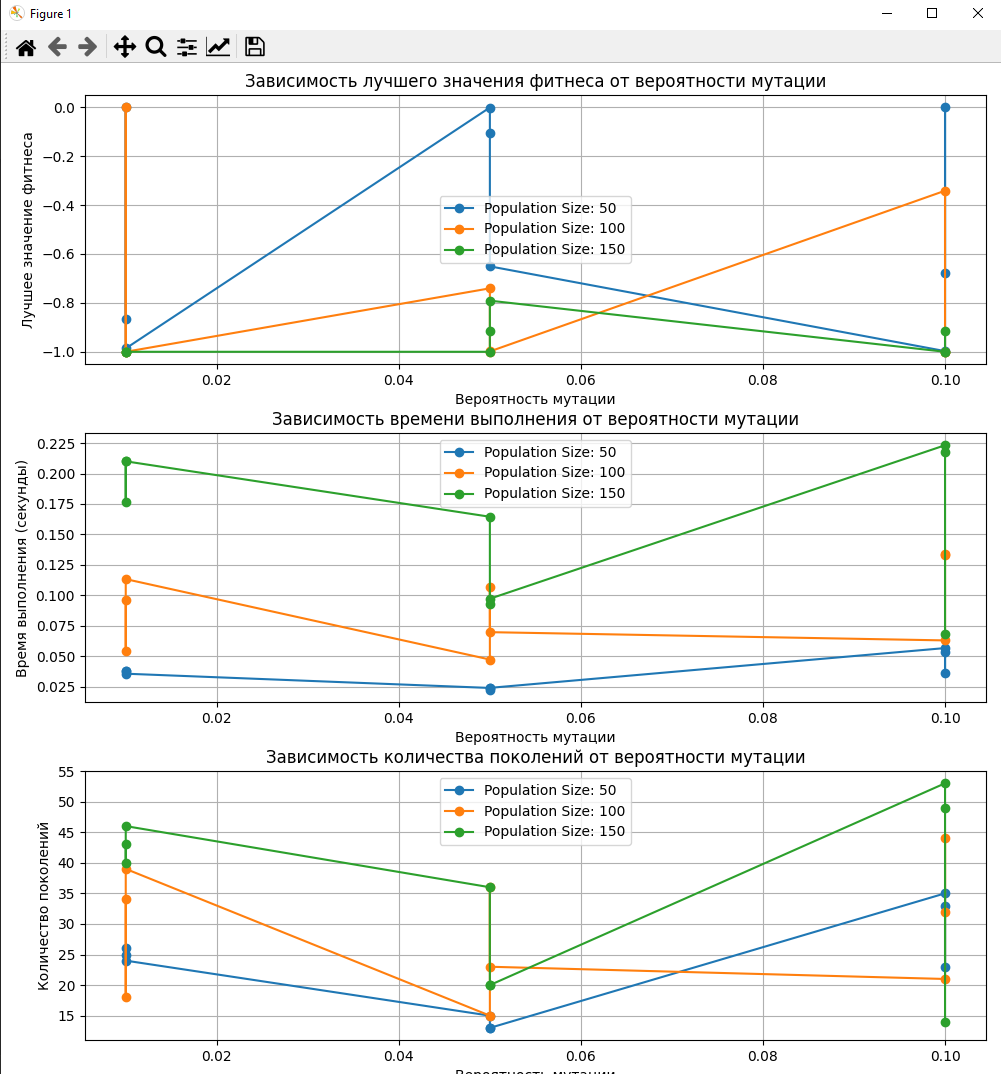
plt.show()

# Результат работы программы



**Результат исследований:**





# Вывод

В ходе выполнения лабораторной работы была произведена модификация представления хромосомы и операторов рекомбинации ГА для оптимизации многомерных функций, а также графическое отображение результатов оптимизации.

Из результатов можно понять, что:

Увеличение размера популяции (числа особей) влечет за собой следующие изменения: время поиска увеличивается, сокращения числа поколений, улучшение точности решения.

Вероятность мутации оказывает влияние следующим образом: время поиска увеличивается, сокращения числа поколений, увеличение шанса нахождения глобального минимума