ГУАП

КАФЕДРА № 43

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

д-р техн. наук, профессор		Скобцов Ю.А.
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ

ПРОСТОЙ ГЕНЕТИЧЕСКИЙ АЛГОРИТМ

по курсу: Эволюционные методы проектирования программно-информационных систем

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. № 4132		Н.И. Карпов
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2024

СОДЕРЖАНИЕ

1 Индивидуальное задание

Вариант 6:

Необходимо найти *максимум* функции: $f(x) = (x-1)\cos(3x-15)$

На интервале: $x \in [-10, 10]$

2 Краткие теоретические сведения

Генетические алгоритмы (ГА) — это один из видов эволюционных алгоритмов, которые используют идеи естественного отбора и генетики для решения оптимизационных задач.

ГА берет множество параметров оптимизационной проблемы и кодирует их последовательностями конечной длины в некотором конечном алфавите (в простейшем случае двоичный алфавит «0» и «1»).

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

1) Оператор репродукции (ОР) – для отбора решений, которые перейдут в следующее поколение;

2) Оператор скрещивания (кроссинговера, ОК) – для генерации новых решений, путем случайного скрещивания генов старых решений;

3) Оператор мутации (ОМ) – для генерации новых решений путём случайного изменения старого.

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.

В каждом поколении множество искусственных особей создается с использованием старых и добавлением новых с хорошими свойствами. Генетические алгоритмы - не просто случайный поиск, они эффективно используют информацию, накопленную в процессе эволюции.

Использующиеся в работе термины:

Ген – элементарный код в хромосоме s_i , называемый также знаком, или детектором (в классическом ГА s_i=0,1).

Хромосома – упорядоченная последовательность генов в виде закодированная структура данных S =(s_1,s_2,…, s_n), определяющая решение.

Популяция – множество особей – потенциальных решений, которые представляются хромосомами.

Поколение – текущая популяция ГА (для текущей итерации алгоритма).

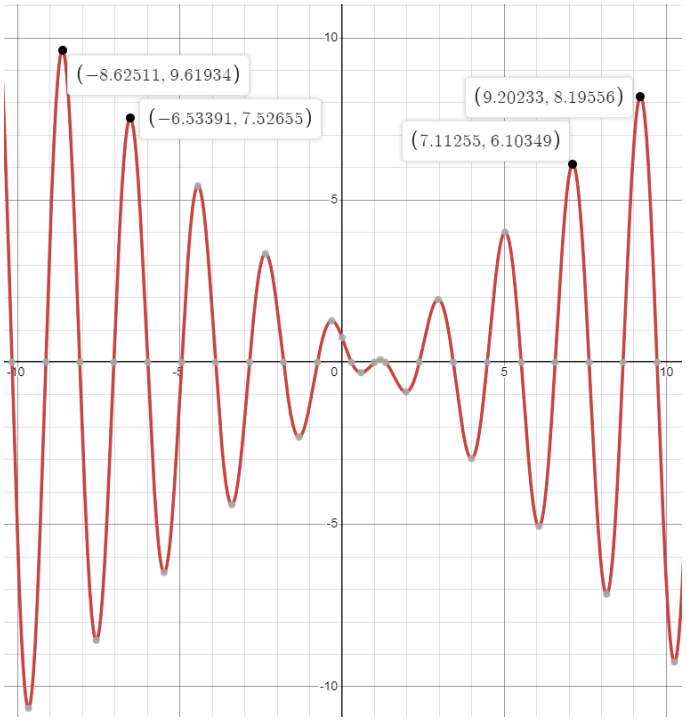
Размер (мощность) популяции N – число особей (решений) в популяции.

Число поколений (генераций) – количество итераций, в течение которых производится генетический поиск.

Фитнесс-функция (полезности) – важнейшее понятие, определяющее меру приспособленности данной особи в популяции. В задачах оптимизации часто представляется целевой функцией или определяет меру близости к оптимальному решению. В обучении может принимать вид функции погрешности (ошибки). На каждой итерации ГА приспособленность каждой особи популяции оценивается с помощью фитнесс-функции.

3 Результаты выполнения работы

Исходная функция имеет следующий вид с максимальным значением в точке (-8.625, 9.619):

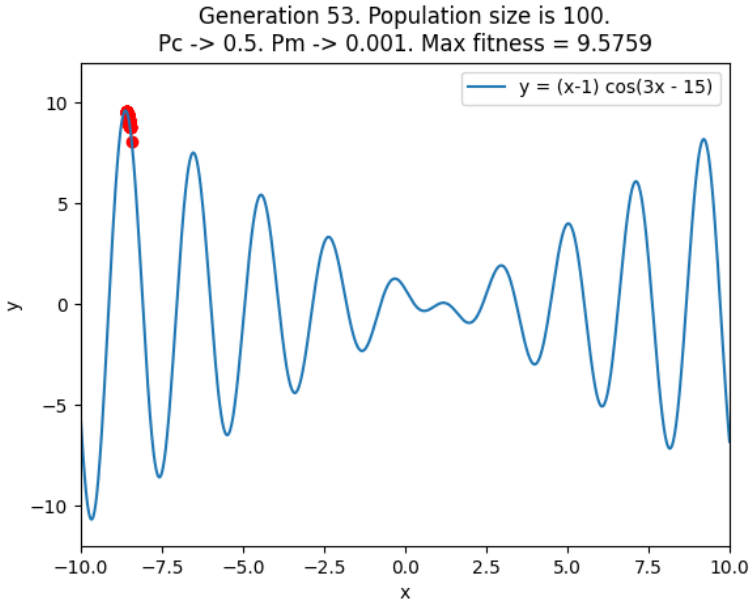


3.1 Исследование решений при разной мощности популяции

Для 100 особей в популяции результаты поиска решения выглядят следующим образом:

```
Generation [50/100]          Fitness delta = 0.057          Max fitness = 9.576          Mean: 9.215
Generation [51/100]          Fitness delta = 0.023          Max fitness = 9.576          Mean: 9.238
Generation [52/100]          Fitness delta = 0.027          Max fitness = 9.576          Mean: 9.265
Generation [53/100]          Fitness delta = -0.004         Max fitness = 9.576          Mean: 9.261

 ================================================================================================
Required time: 7.69s. Found answer: 9.575910.  Required generations: 54. Fitness mean delta: 0.004016
```

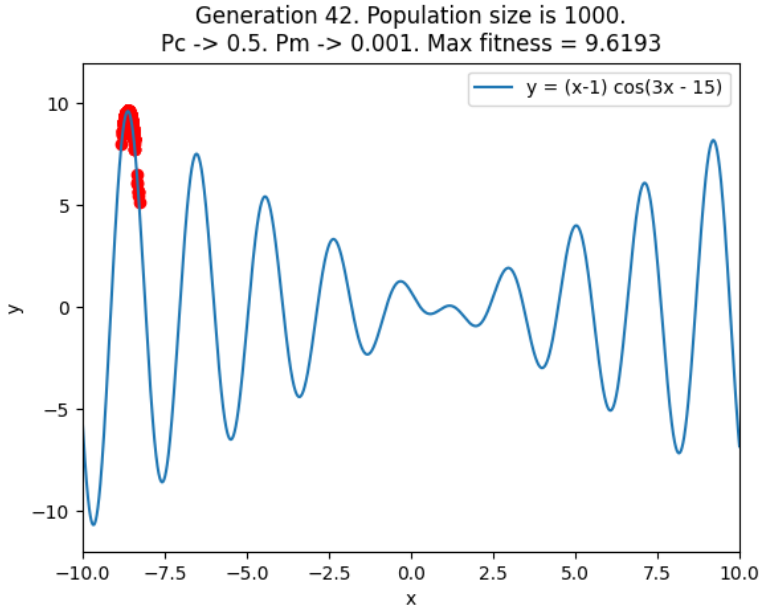
Т.е. на поиск решения ушло 7,69 секунд и 54 поколения, точность ответа составила $9{,}619 - 9{,}576 \approx \mathbf{0,043}$.



Для 1000 особей:

```
Generation [39/100]          Fitness delta = 0.158          Max fitness = 9.619          Mean: 9.207
Generation [40/100]          Fitness delta = 0.024          Max fitness = 9.619          Mean: 9.231
Generation [41/100]          Fitness delta = 0.096          Max fitness = 9.619          Mean: 9.327
Generation [42/100]          Fitness delta = -0.004         Max fitness = 9.619          Mean: 9.323

 ================================================================================================
Required time: 9.39s. Found answer: 9.619338.  Required generations: 43. Fitness mean delta: 0.004126

F:\Coding\py\evolutionary-programming\simpleGeneticAlgorithm>
```

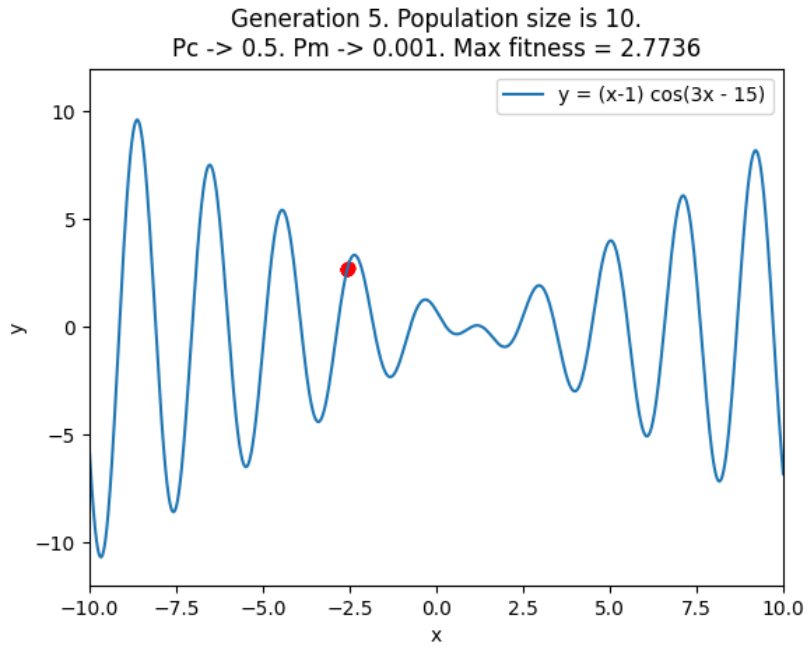
Т.е. на поиск решения ушло 9,39 секунд и 42 поколения, точность ответа составила $9{,}61934 - 9{,}61933 \approx \mathbf{0,00001}$.



Для 10 особей:

```
Generation [1/100]          Fitness delta = -1.658          Max fitness = 2.715          Mean: -1.658

Generation [2/100]          Fitness delta = 2.132           Max fitness = 2.768          Mean: 0.474

Generation [3/100]          Fitness delta = 1.056           Max fitness = 2.768          Mean: 1.530

Generation [4/100]          Fitness delta = 1.201           Max fitness = 2.768          Mean: 2.731

Generation [5/100]          Fitness delta = -0.000          Max fitness = 2.774          Mean: 2.731


 ========================================================================================================
Required time: 0.99s. Found answer: 2.773647.  Required generations: 6. Fitness mean delta: 0.000033
```

Т.е. на поиск решения ушло 0,99 секунды и 5 поколений, точность ответа составила $9{,}619 - 2{,}774 \approx \mathbf{6{,}845}$.



Следовательно, мощность популяции значительно влияет на точность ответа. При большом размере популяции более вероятно получение точного ответа. На слишком малых популяциях вероятно попадание в локальный экстремум (что снижает точность).

С другой стороны, количество особей влияет на скорость выполнения алгоритма. Чем больше размер популяции, тем больше затрат по времени и памяти.

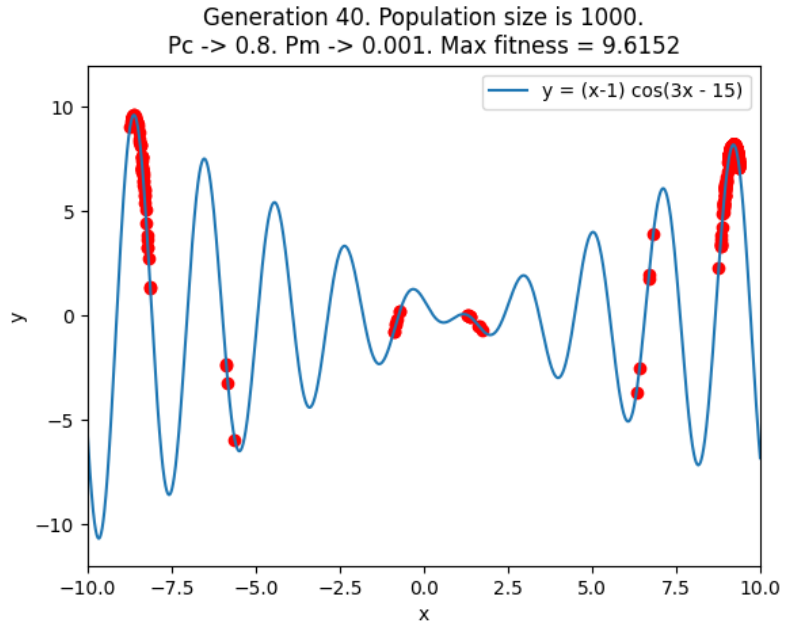
3.2 Исследование решений при разной вероятности кроссинговера

Для вероятности кроссинговера в 0,5 поиск решения уже осуществлялся ранее.

Для вероятности кроссинговера в 0,8:

```
Generation [38/100]          Fitness delta = -0.009          Max fitness = 9.612          Mean: 7.575
Generation [39/100]          Fitness delta = 0.065           Max fitness = 9.615          Mean: 7.641
Generation [40/100]          Fitness delta = 0.000           Max fitness = 9.615          Mean: 7.641

 ============================================================================================
Required time: 8.90s. Found answer: 9.615153.  Required generations: 41. Fitness mean delta: 0.000269
```

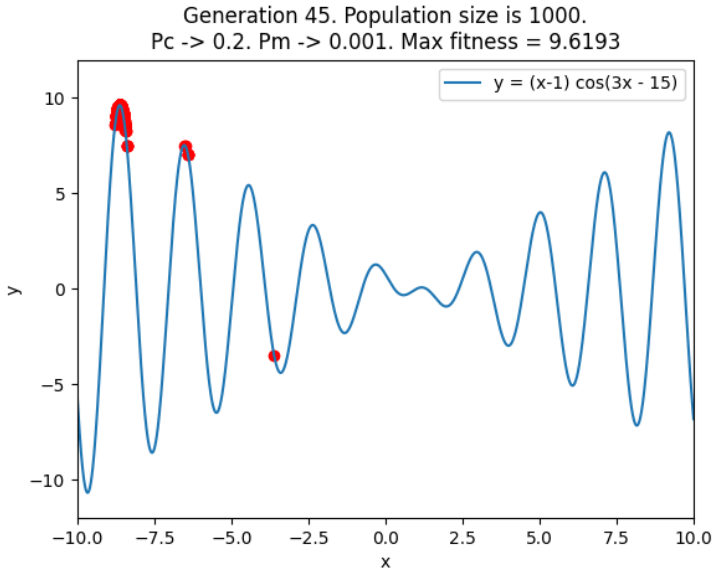
Т.е. поиск решения ускорился, но его точность – уменьшилась. Это связано с тем, что в ходе работы алгоритма появилось большее разнообразие генов, что ускорило поиск нахождения новых решений, но снизило устойчивость старых решений. Поэтому на графике существует несколько устойчивых областей, вместо одной наиболее подходящей:



Для вероятности кроссинговера в 0,2:

```
Generation [42/100]            Fitness delta = 0.021          Max fitness = 9.619            Mean: 9.314
Generation [43/100]            Fitness delta = 0.051          Max fitness = 9.619            Mean: 9.365
Generation [44/100]            Fitness delta = 0.058          Max fitness = 9.619            Mean: 9.423
Generation [45/100]            Fitness delta = 0.005          Max fitness = 9.619            Mean: 9.427

 ================================================================================================
Required time: 10.05s. Found answer: 9.619341.  Required generations: 46. Fitness mean delta: 0.004635
```

Т.е. поиск решения замедлился, но его точность – увеличилась. Это аналогично связано с тем, что старые и хорошие по точности ответы стали более стабильны. В таких условиях труднее найти новые решения и существует риск остаться в локальном экстремуме.



Следовательно, наиболее оптимальной вероятностью кроссинговера является вероятность около 50%. Она обеспечивает достаточною скорость эволюции и разнообразие генов, но снижает вероятность попадания в локальный экстремум или нахождение недостаточно точного решения.

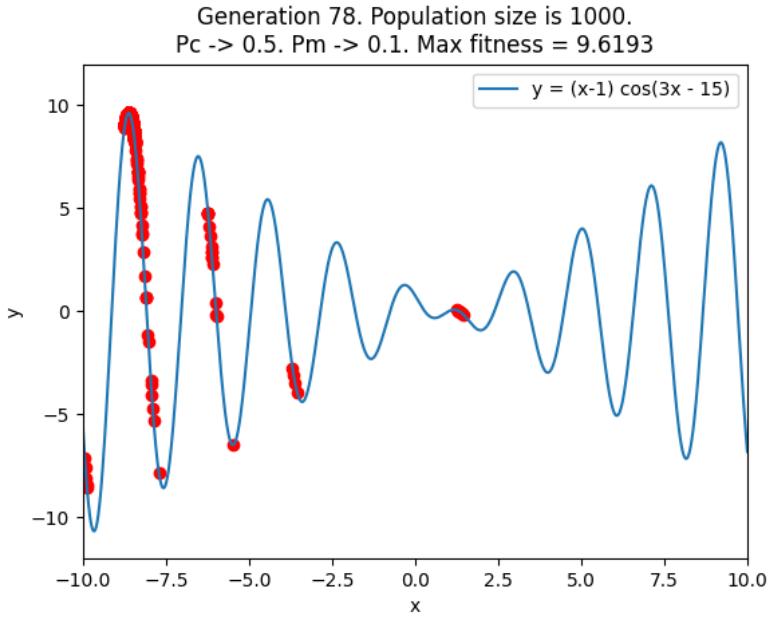
3.3 Исследование решений при разной вероятности мутации

Для вероятности мутации в 0,001 поиск решения уже осуществлялся ранее.

Для вероятности мутации в 0,1:

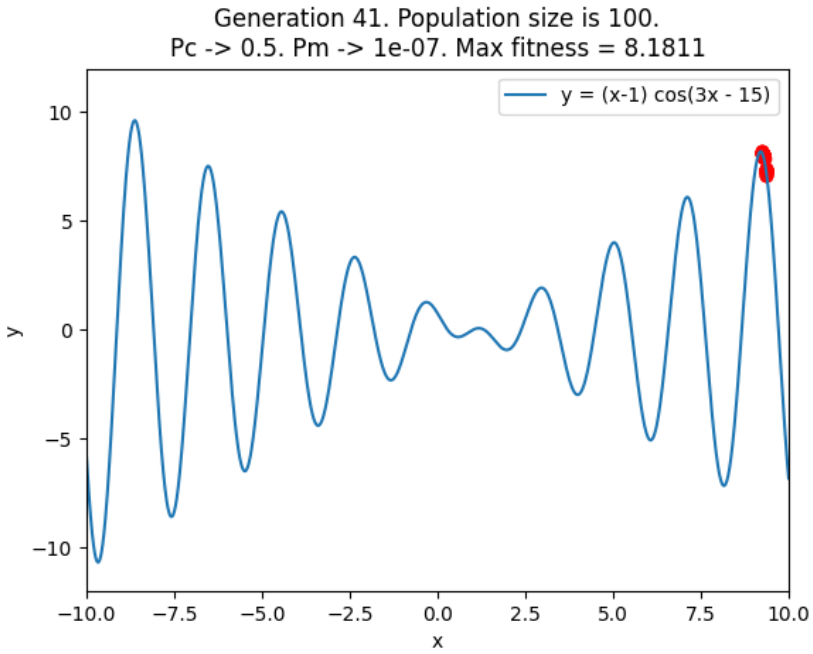
```
Generation [76/100]          Fitness delta = 0.058          Max fitness = 9.619          Mean: 8.629
Generation [77/100]          Fitness delta = 0.031          Max fitness = 9.619          Mean: 8.660
Generation [78/100]          Fitness delta = 0.005          Max fitness = 9.619          Mean: 8.664

 ============================================================================================================
Required time: 16.97s. Found answer: 9.619321.  Required generations: 79. Fitness mean delta: 0.004527
```

Поиск решения оказывается крайне долгим и сложным (малая скорость сходимости). Сами решения остаются крайне неустойчивыми.



Для вероятности мутации в 1e-7 увеличивается вероятность остаться в локальном экстремуме:

```
Generation [39/100]          Fitness delta = 0.008          Max fitness = 8.184          Mean: 7.905
Generation [40/100]          Fitness delta = 0.017          Max fitness = 8.184          Mean: 7.923
Generation [41/100]          Fitness delta = 0.002          Max fitness = 8.181          Mean: 7.925

 ============================================================================================================
Required time: 5.98s. Found answer: 8.181089.  Required generations: 42. Fitness mean delta: 0.002176
```

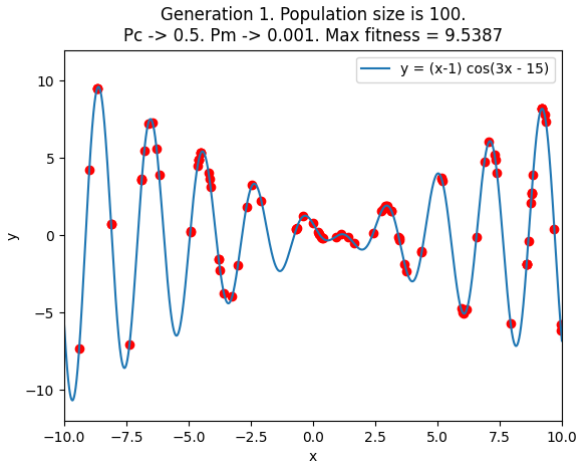


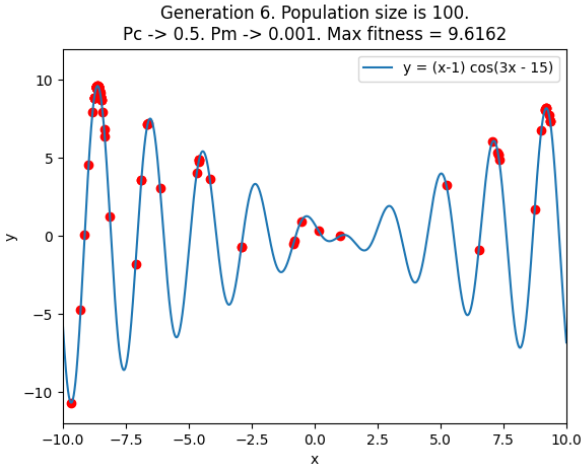
Следовательно, наиболее оптимальной вероятностью мутации является вероятность около 0.1%-1%. С такой вероятностью остаётся больше шансов избежать локального экстремума, при этом сходимость решения остаётся быстрой.

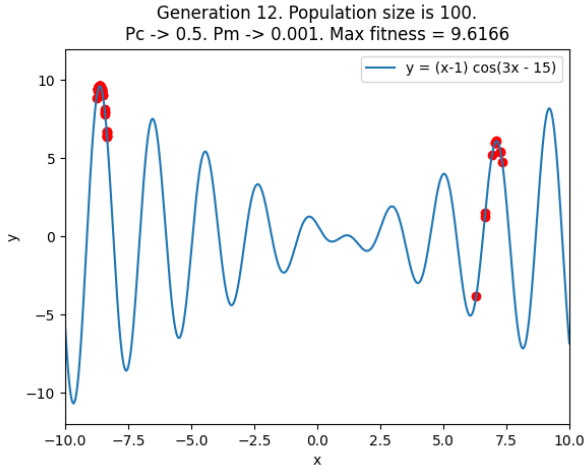
3.4 Поиск решений

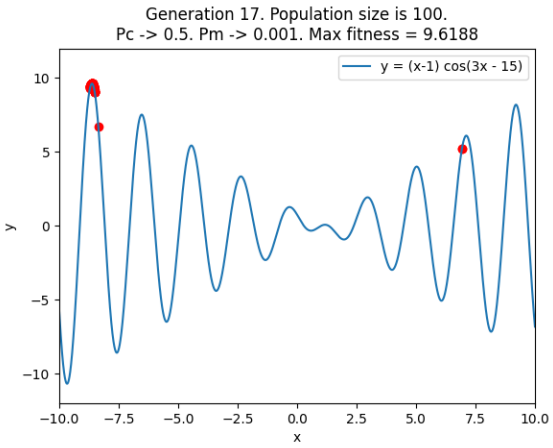
Таким образом, самыми эффективными параметрами оказались популяция в 100-1000 особей, вероятность кроссинговера порядка 50% и вероятность мутации около 0,1%.

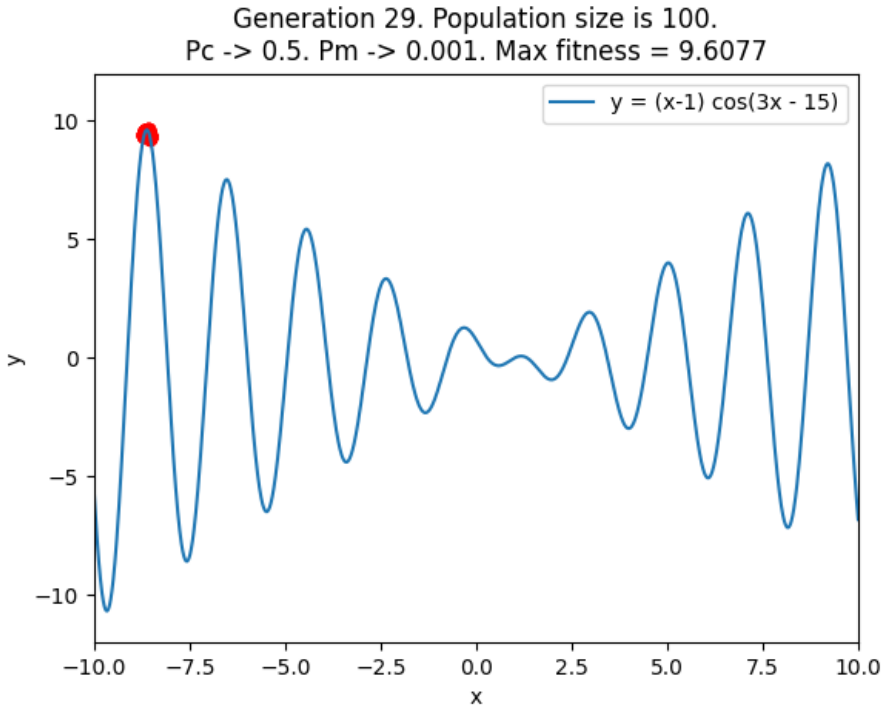
Поиск решения с такой конфигурацией будет выглядеть следующим образом:


Generation 1. Population size is 100.
Pc -> 0.5. Pm -> 0.001. Max fitness = 9.5387
y = (x-1) cos(3x - 15)
x
y


Generation 6. Population size is 100.
Pc -> 0.5. Pm -> 0.001. Max fitness = 9.6162
y = (x-1) cos(3x - 15)
x
y


Generation 12. Population size is 100.
Pc -> 0.5. Pm -> 0.001. Max fitness = 9.6166
y = (x-1) cos(3x - 15)
x
y


Generation 17. Population size is 100.
Pc -> 0.5. Pm -> 0.001. Max fitness = 9.6188
y = (x-1) cos(3x - 15)
x
y


Generation 29. Population size is 100.
Pc -> 0.5. Pm -> 0.001. Max fitness = 9.6077
y = (x-1) cos(3x - 15)
x
y

На данных графиках видно, как алгоритм эффективно использует информацию, накапливаемую в процессе эволюции. Это проявляется в постепенной концентрации решений вокруг экстремумов.

3.5 Листинг программы

```
from math import cos
import genalg

POPULATION_SIZE = 100
CROSSING_OVER_PROBABILITY = 0.5
MUTATION_PROBABILITY = 0.001
MAX_GENERATIONS = 100
EPSILON = 0.0005

LEFT_EDGE, RIGHT_EDGE = -10, 10


# Function is (x-1) cos(3x - 15)
def fitness_function(chromosome):
    return (chromosome - 1) * cos(3 * chromosome - 15)


if __name__ == '__main__':
    genetic_optimizer = genalg.SimpleGeneticAlgorithm(fitness_function,
POPULATION_SIZE, MAX_GENERATIONS,
                                                      CROSS-
ING_OVER_PROBABILITY, MUTATION_PROBABILITY,
                                                      LEFT_EDGE,
RIGHT_EDGE, EPSILON)

    genetic_optimizer.start()



import matplotlib.pyplot as plt
import numpy as np

from time import time
from random import randint, random

UNSIGNED_INT_BITS_NUMBER = 16


def decode_number(segment_number: int, left_interval, right_interval) ->
float:
    """
    Decode an int number specified segment on [left_interval,
right_interval] into float number from this interval.
    """
    # x = b - x' * (a - b) / segments_number
    return left_interval + segment_number * (right_interval -
left_interval) / (2 ** UNSIGNED_INT_BITS_NUMBER - 1)


def swap_bits(first_chromosome, second_chromosome, k):
    """
    Swaps bits between two chromosomes from k bit's position.
    :param first_chromosome: first number to be swapped
```

```
    :param second_chromosome: second number to be swapped
    :param k: bit's position to start swap
    :return: inverted number
    """
    # Create strings with a binary representations of numbers.
    first_chromosome_original = bin(first_chromosome)[2:].zfill(16)
    second_chromosome_original = bin(second_chromosome)[2:].zfill(16)

    # Swap bits from k
    swapped_first_chromosome =
f'0b{first_chromosome_original[:k]}{second_chromosome_original[k:]}'
    swapped_second_chromosome =
f'0b{second_chromosome_original[:k]}{first_chromosome_original[k:]}'

    return int(swapped_first_chromosome, 2), int(swapped_second_chromosome,
2)


def invert_bit(chromosome, k):
    """
    Inverts one bit from k position on 16-bits number.
    :param chromosome: number to be inverted
    :param k: bit's position to invert
    :return: inverted number
    """
    # Create a string with a binary representation of a number.
    original = bin(chromosome)[2:].zfill(16)
    inverted = original[:k - 1]

    if original[k - 1] == '0':
        inverted += '1'
    else:
        inverted += '0'

    inverted += original[k:]

    return int(f'0b{inverted}', 2)


class SimpleGeneticAlgorithm:
    """
    Solves task of finding maximum of target function (fitness function)
using genetic algorithm.
    :param fitness_function: target function
    :param population_size: size of the population
    :param max_generations: number of max available generations. For a sit-
uation when it is impossible to find a satisfying optimum
    :param crossover_probability: probability of a crossover
    :param mutation_probability: probability of a mutation
    :param epsilon: stop algorithm when  mean fitness function is less than
or equal to this value
    """

    def __init__(self, fitness_function, population_size, max_generations,
                 crossover_probability, mutation_probability, left_edge,
right_edge, epsilon=0.05):
        self.fitness_function = fitness_function
        self.population_size = population_size
        self.max_generations = max_generations
        self.crossover_probability = crossover_probability
        self.mutation_probability = mutation_probability
```

```
        self.left_edge = left_edge
        self.right_edge = right_edge
        self.epsilon = epsilon

        self.population = []
        self.fitness_values = np.zeros(population_size, float)

        for i in range(population_size):
            # Decode with 16-bits number.
            self.population.append(randint(0, 65535))

    def start(self):
        generation = 1  # generations counter
        last_fitness_mean = 0.  # mean fitness
        mean_fitness_delta = 1.  # delta between current and previous mean
fitness function value

        plt.ion()
        start_time = time()

        while generation < self.max_generations and abs(mean_fitness_delta)
> self.epsilon:
            # 1. Evaluate fitness.
            self.evaluate()

            # 2. Reproduction.
            self.reproduction()

            # 3. Crossing.
            self.crossover()

            # 4. Mutate.
            self.mutate()

            mean_fitness_delta = self.fitness_values.mean() -
last_fitness_mean
            last_fitness_mean = self.fitness_values.mean()

            print(f'Generation [{generation}/{self.max_generations}]\t\t' +
                  f'Fitness delta = {mean_fitness_delta:.3f}\t\tMax fitness
= {max(self.fitness_values):.3f}\t\t' +
                  f'Mean: {(sum(self.fitness_values) /
self.population_size):.3f}\n')

            self.draw_plot(generation)

            generation += 1

        end_time = time()
        plt.show(block=True)

        print('\n', '=' * 100)
        print(f'Required time: {end_time - start_time:.2f}s. Found answer:
{max(self.fitness_values):4f}. ',
              f'Required generations: {generation}. Fitness mean delta:
{abs(mean_fitness_delta):3f}')

    def evaluate(self):
        """
        Evaluates the fitness of the population.
        """
```

```
        for i in range(self.population_size):
            # Count fitness of every chromosome in population
            self.fitness_values[i] =
self.fitness_function(decode_number(self.population[i],

self.left_edge,

self.right_edge)
                                                                    )

    def reproduction(self):
        """
        Roulette based reproduction algorithm.
        """
        fitness_copy = []

        # Normalize to avoid problems with negative numbers.
        min_fitness = min(self.fitness_values)
        for fitness in self.fitness_values:
            if min_fitness < 0:
                fitness_copy.append(fitness + abs(min_fitness) +
0.0000000001)
            else:
                fitness_copy.append(fitness)

        total_fitness = sum(fitness_copy)
        probabilities = [fitness / total_fitness for fitness in fit-
ness_copy]
        new_population = []

        for i in range(self.population_size):
            current_wheel_probability = 0.
            random_probability = random()

            # Emulate roulette pass.
            for j in range(self.population_size):
                current_wheel_probability += probabilities[j]

                if random_probability < current_wheel_probability:
                    # Random chosen probability in range
[last_chromosome_p, new_chromosome_p).
                    new_population.append(self.population[j])
                    break

            # If there is case with rounding error (last chromosome wasn't
added) add last chromosome manually.
            if i != new_population.__len__() - 1:
                new_population.append(self.population[-1])

        # Update population.
        for i in range(self.population_size):
            self.population[i] = new_population[i]

    def crossover(self):
        """
        Randomly distributes population on pairs and makes crossing (with
probability).
        """
        free_chromosomes = self.population.copy()
        pairs = []
```

```
        # While there is free chromosomes to make pair.
        while len(free_chromosomes) > 1:
            pair = []

            # Choose two random elements.
            for i in range(2):
                rand_index = randint(0, len(free_chromosomes) - 1)
                pair.append(free_chromosomes.pop(rand_index))

            pairs.append(pair)

        for pair in pairs:
            # With probability.
            if random() < self.crossover_probability:
                cross_dot = randint(1, UNSIGNED_INT_BITS_NUMBER - 1)

                first_index = self.population.index(pair.pop())
                second_index = self.population.index(pair.pop())

                first_swapped, second_swapped =
swap_bits(self.population[first_index],

self.population[second_index],
                                                                    cross_dot)

                self.population[first_index], self.population[second_index]
= first_swapped, second_swapped

    def mutate(self):
        for i in range(self.population_size):
            if random() < self.mutation_probability:
                rand_index = randint(1, UNSIGNED_INT_BITS_NUMBER)
                self.population[i] = invert_bit(self.population[i],
rand_index)

    def draw_plot(self, generation):
        plt.clf()

        x = [decode_number(i, self.left_edge, self.right_edge) for i in
range(0, 65536, 10)]
        y = [self.fitness_function(value) for value in x]
        plt.plot(x, y, label='y = (x-1) cos(3x - 15)')

        decoded_dots = []
        decoded_dots_y = []
        for dot in self.population:
            decoded_dots.append(decode_number(dot, self.left_edge,
self.right_edge))
            decoded_dots_y.append(self.fitness_function(decode_number(dot,
self.left_edge, self.right_edge)))

        plt.scatter(decoded_dots, decoded_dots_y, color='red')

        plt.xlim(self.left_edge, self.right_edge)
        plt.ylim(-12, 12)
        plt.title(f"Generation {generation}. Population size is
{self.population_size}.\n" +
                  f"Pc -> {self.crossover_probability}. Pm ->
{self.mutation_probability}. " +
                  f"Max fitness = {max(self.fitness_values):.4f}")
        plt.xlabel('x')
```

```
plt.ylabel('y')
plt.legend()

plt.draw()
plt.pause(0.1)
```

4 Ответ на контрольный вопрос

Опишите 1-точечный оператор кроссинговера (ОК) и приведите пример его работы.

1-точечный оператор кроссинговера — это один из методов скрещивания (кроссинговера) в генетических алгоритмах. Этот метод заключается в том, что два родителя обмениваются генетическим материалом, разделяясь в одном случайно выбранном месте (точке кроссинговера), после чего их части комбинируются для создания двух потомков.

Выбор точки кроссинговера осуществляется случайным образом. В результате создаются два новых потомка, которые комбинируют части генетического материала от двух родителей.

Пример:
Пусть, есть два родителя с хромосомами длиной 8 бит:
Родитель 1: 10111010
Родитель 2: 01100111

Пусть выбирается случайная точка k = 4

	1	2	3	4	5	6	7	8
Родитель 1:	1	0	1	1	| 1	0	1	0
Родитель 2:	0	1	1	0	| 0	1	1	1

Тогда:
Потомок 1: 1011 | 0111 → 10110111
Потомок 2: 0110 | 1010 → 01101010

5 Вывод

В ходе работы была создана программа для решения оптимизационной задачи и установлено, что генетические алгоритмы являются хорошим инструментом для задач такого рода.