

ГУАП

КАФЕДРА № 43

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ
ПРЕПОДАВАТЕЛЬ

д-р техн. наук, профессор

должность, уч. степень, звание

подпись, дата

Скобцов Ю.А.

инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ

Оптимизация многомерных функций с помощью ГА

по курсу: Эволюционные методы проектирования программно-информационных систем

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. № 4132

подпись, дата

Н.И. Карпов

инициалы, фамилия

Санкт-Петербург 2024

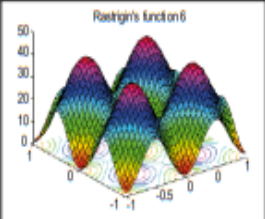
СОДЕРЖАНИЕ

1	Индивидуальное задание	3
2	Краткие теоретические сведения	3
3	Результаты выполнения работы	4
3.1	Исследование решений при $n=2$	4
3.2	Исследование решений при $n>2$	5
3.3	Исследование решений при разных параметрах	6
3.4	Листинг программы	7
4	Вывод	13

1 Индивидуальное задание

Вариант 6:

Необходимо найти минимум многомерной функции с использованием вещественного представления хромосом в генетических алгоритмах.

6	Rastrigin's function 6	global minimum $f(x)=0; x(i)=0,$ $i=1:n.$	$f_6(x) = 10 \cdot n + \sum_{i=1}^{n-1} (x_i^2 - 10 \cdot \cos(2 \cdot \pi \cdot x_i))$ $-5.12 \leq x_i \leq 5.12$ $f6(x) = 10 \cdot n + \text{sum}(x(i)^2 - 10 \cdot \cos(2 \cdot \pi \cdot x(i))),$ $i=1:n;$	
---	------------------------	---	--	---

2 Краткие теоретические сведения

При работе с оптимизационными задачами в непрерывных пространствах вполне естественно представлять гены напрямую вещественными числами. В этом случае хромосома есть вектор вещественных чисел. Их точность будет определяться исключительно разрядной сеткой той ЭВМ, на которой реализуется real-coded алгоритм. Длина хромосомы будет совпадать с длиной вектора-решения оптимизационной задачи, иначе говоря, каждый ген будет отвечать за одну переменную. Генотип объекта становится идентичным его фенотипу. Вышесказанное определяет список основных преимуществ real-coded алгоритмов:

1. Использование непрерывных генов делает возможным поиск в больших пространствах (даже в неизвестных), что трудно делать в случае двоичных генов, когда увеличение пространства поиска сокращает точность решения при неизменной длине хромосомы.

2. Одной из важных черт непрерывных ГА является их способность к локальной настройке решений.

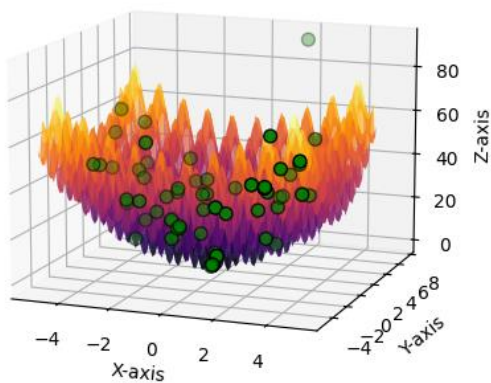
3. Использование RGA для представления решений удобно, поскольку близко к постановке большинства прикладных задач. Кроме того, отсутствие операций кодирования/декодирования, которые необходимы в BGA, повышает скорость работы алгоритма. Появление новых особей в популяции канонического ГА обеспечивают несколько биологических операторов: отбор, скрещивание и мутация. В качестве операторов отбора особей в родительскую пару здесь подходят любые известные из BGA: рулетка, турнирный, случайный. Однако операторы

скрещивания и мутации в классических реализациях работают с битовыми строками. Необходимы реализации, учитывающие специфику realcoded алгоритмов. Также рекомендуется использовать стратегию элитизма – лучшая особь сохраняется отдельно и не стирается при смене эпох, принимая при этом участие в отборе и рекомбинации.

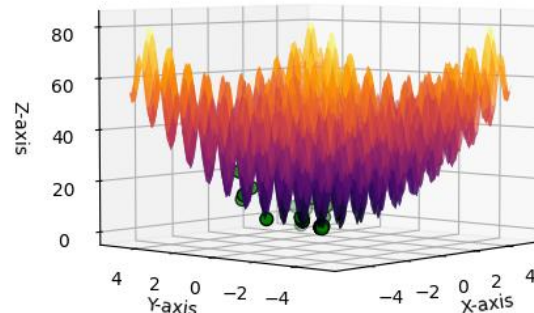
3 Результаты выполнения работы

3.1 Исследование решений при $n=2$

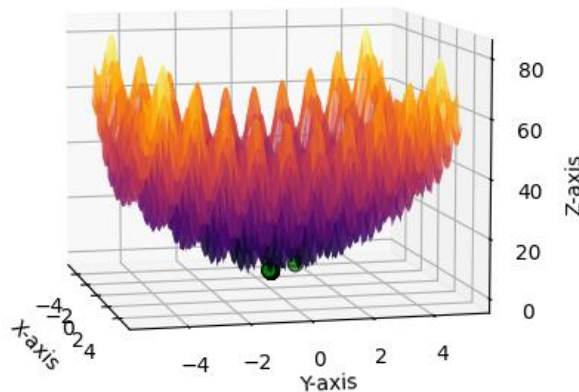
Generation 1. Population size is 100.
Pc -> 0.5. Pm -> 0.001. Min fitness = 0.7377



Generation 2. Population size is 100.
Pc -> 0.5. Pm -> 0.001. Min fitness = 0.7377



Generation 4. Population size is 100.
Pc -> 0.5. Pm -> 0.001. Min fitness = 1.0089



Generation [45/50]	Fitness delta = 0.000	Min fitness = 0.104	Mean: 0.122
Generation [46/50]	Fitness delta = 0.000	Min fitness = 0.096	Mean: 0.122
Generation [47/50]	Fitness delta = 0.008	Min fitness = 0.099	Mean: 0.130
Generation [48/50]	Fitness delta = -0.013	Min fitness = 0.099	Mean: 0.117
Generation [49/50]	Fitness delta = 0.000	Min fitness = 0.106	Mean: 0.117

```

=====
Required time: 0.14s. Found answer: 0.105920. Required generations: 50. Fitness mean delta: 0.000056
F:\Coding\py\modeling\realGeneticAlgorithm>

```

Т.е. на поиск решения ушло 0,14 секунд и 50 поколений.

Похожий результат был достигнут средствами библиотеки GEAN:

```

48 63      0.393092    0.144449
49 59      0.595292    0.144447
50 57      0.43918     0.144446
Best individual is [-8.498808857685091e-05, 0.027015104153140096], with fitness: 0.14444556837065647

```

3.2 Исследование решений при $n > 2$

При $n = 3$

Generation [45/50]	Fitness delta = 0.639	Min fitness = 1.104	Mean: 5.871
Generation [46/50]	Fitness delta = -0.599	Min fitness = 1.099	Mean: 5.272
Generation [47/50]	Fitness delta = -1.530	Min fitness = 0.365	Mean: 3.741
Generation [48/50]	Fitness delta = -0.912	Min fitness = 0.124	Mean: 2.829
Generation [49/50]	Fitness delta = 0.449	Min fitness = 0.124	Mean: 3.279

```

=====
Required time: 0.17s. Found answer: 0.124081. Required generations: 50. Fitness mean delta: 0.449346
n = 3, Pc = 0.7, Pm = 0.1

```

Т.е. на поиск решения ушло 0,16 секунд и 50 поколений.

При $n = 4$

Generation [44/50]	Fitness delta = -0.021	Min fitness = 0.101	Mean: 0.305
Generation [45/50]	Fitness delta = -0.089	Min fitness = 0.102	Mean: 0.216
Generation [46/50]	Fitness delta = 0.121	Min fitness = 0.102	Mean: 0.337
Generation [47/50]	Fitness delta = -0.110	Min fitness = 0.100	Mean: 0.227
Generation [48/50]	Fitness delta = 0.006	Min fitness = 0.102	Mean: 0.233
Generation [49/50]	Fitness delta = 0.050	Min fitness = 0.103	Mean: 0.283

```

=====
Required time: 0.17s. Found answer: 0.103291. Required generations: 50. Fitness mean delta: 0.049681
n = 4, Pc = 0.7, Pm = 0.1

```

Т.е. на поиск решения ушло 0,17 секунд и 50 поколений.

При $n = 5$

Generation [45/50]	Fitness delta = -0.292	Min fitness = 0.814	Mean: 6.291
Generation [46/50]	Fitness delta = -2.866	Min fitness = 0.814	Mean: 3.425
Generation [47/50]	Fitness delta = 0.836	Min fitness = 0.862	Mean: 4.260
Generation [48/50]	Fitness delta = -0.301	Min fitness = 0.699	Mean: 3.959
Generation [49/50]	Fitness delta = -0.642	Min fitness = 0.873	Mean: 3.317

Required time: 0.16s. Found answer: 0.872608. Required generations: 50. Fitness mean delta: 0.641755
n = 5, Pc = 0.7, Pm = 0.1

Т.е. на поиск решения ушло 0,16 секунд и 50 поколений.

Точность решения понижается с увеличением количества измерений, скорость значений меняется незначительно. Для компенсации уменьшения точности можно увеличить вероятность мутации, количество поколений или размер популяции.

3.3 Исследование решений при разных параметрах

В ходе работы было установлено, что размер популяции положительно влияет на точность, но негативно влияет на скорость вычислений:

Размер популяции 1000:

Generation [46/50]	Fitness delta = -0.003	Min fitness = 0.000	Mean: 0.101
Generation [47/50]	Fitness delta = -0.004	Min fitness = 0.000	Mean: 0.097
Generation [48/50]	Fitness delta = 0.006	Min fitness = 0.000	Mean: 0.103
Generation [49/50]	Fitness delta = -0.013	Min fitness = 0.000	Mean: 0.090

Required time: 4.38s. Found answer: 0.000000. Required generations: 50. Fitness mean delta: 0.013001
n = 3, Pc = 0.7, Pm = 0.1

Размер популяции 10:

Generation [47/50]	Fitness delta = 0.259	Min fitness = 10.210	Mean: 11.446
Generation [48/50]	Fitness delta = 3.096	Min fitness = 10.277	Mean: 14.542
Generation [49/50]	Fitness delta = 1.192	Min fitness = 10.277	Mean: 15.734

Required time: 0.02s. Found answer: 10.276817. Required generations: 50. Fitness mean delta: 1.191685
n = 3, Pc = 0.7, Pm = 0.1

Большая вероятность кроссинговера или мутации уменьшает шанс попасть в локальный минимум, но ухудшает сходимость:

Generation [47/50]	Fitness delta = -0.454	Min fitness = 0.032	Mean: 1.351
Generation [48/50]	Fitness delta = -0.687	Min fitness = 0.032	Mean: 0.664
Generation [49/50]	Fitness delta = -0.102	Min fitness = 0.031	Mean: 0.562
=====			
Required time: 0.16s. Found answer: 0.031365. Required generations: 50. Fitness mean delta: 0.102306 n = 3, Pc = 0.7, Pm = 0.5			
Generation [46/50]	Fitness delta = -0.016	Min fitness = 7.079	Mean: 7.231
Generation [47/50]	Fitness delta = -0.001	Min fitness = 7.092	Mean: 7.230
Generation [48/50]	Fitness delta = -0.011	Min fitness = 7.030	Mean: 7.219
Generation [49/50]	Fitness delta = -0.004	Min fitness = 7.070	Mean: 7.215
=====			
Required time: 0.17s. Found answer: 7.069543. Required generations: 50. Fitness mean delta: 0.004079 n = 3, Pc = 0.7, Pm = 0.0001			
Generation [46/50]	Fitness delta = 0.081	Min fitness = 0.032	Mean: 0.175
Generation [47/50]	Fitness delta = -0.050	Min fitness = 0.032	Mean: 0.125
Generation [48/50]	Fitness delta = 0.021	Min fitness = 0.033	Mean: 0.146
Generation [49/50]	Fitness delta = 0.001	Min fitness = 0.032	Mean: 0.147
=====			
Required time: 0.14s. Found answer: 0.031927. Required generations: 50. Fitness mean delta: 0.001052 n = 3, Pc = 0.3, Pm = 0.1			
Generation [46/50]	Fitness delta = 0.036	Min fitness = 0.019	Mean: 0.147
Generation [47/50]	Fitness delta = -0.041	Min fitness = 0.019	Mean: 0.106
Generation [48/50]	Fitness delta = 0.048	Min fitness = 0.019	Mean: 0.154
Generation [49/50]	Fitness delta = -0.040	Min fitness = 0.019	Mean: 0.114
=====			
Required time: 0.17s. Found answer: 0.019124. Required generations: 50. Fitness mean delta: 0.040005 n = 3, Pc = 0.9, Pm = 0.1			

3.4 Листинг программы

```
import numpy as np
import genalg

POPULATION_SIZE = 100
CROSSING_OVER_PROBABILITY = 0.9
MUTATION_PROBABILITY = 0.1
MAX_GENERATIONS = 50
EPSILON = 0.
N = 3

LEFT_EDGE, RIGHT_EDGE = -5.12, 5.12

# Function is Rastrigin's function 6
```

```

def fitness_function(chromosome):
    return 10 * len(chromosome) + np.sum(np.power(chromosome, 2) - 10 *
np.cos(2 * np.pi * chromosome))

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

    genetic_optimizer.start()

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

from time import time
from random import randint, random

def count_mutation_delta(generation, total_generations, limit, b):
    # Random number [0, 1)
    r = random()

    degree = (1 - generation / total_generations) ** b
    return limit ** (1 - r ** degree)

class RealGeneticAlgorithm:
    """
    Solves task of finding maximum of target function (fitness function)
    using genetic algorithm.
    :param fitness_function: target function
    :param n: number of dimensions
    :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, n: int, population_size: int,
max_generations: int,
                    crossover_probability: float,
                    mutation_probability: float, left_edge: float, right_edge:
float, 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.n = n

```



```

        # Every chromosome is vector with n elements (xi). Initially uni-
        formly randomized.
        self.population = np.random.uniform(left_edge, right_edge,
size=(population_size, n))
        self.fitness_values = np.zeros(population_size, float)

    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

        start_time = time()

        while generation < self.max_generations:
            # 1. Evaluate fitness.
            self.evaluate()

            # 2. Reproduction.
            self.reproduction()

            # 3. Crossing.
            self.crossover()

            # 4. Mutate.
            self.mutate(generation, 1., 1., 0.8)

            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\tMin fitness
= {min(self.fitness_values):.3f}\t\t' +
                f'Mean: {(sum(self.fitness_values) /
self.population_size):.3f}\n')

            # if self.n == 2:
            #     self.evaluate()
            #     self.draw_plot(generation)

            generation += 1

        end_time = time()

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

    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(self.population[i])

```

```

def reproduction(self):
    """
    Roulette based reproduction algorithm.
    """
    fitness_copy = self.fitness_values.copy()
    fitness_copy_inverted = 1 / fitness_copy

    total_fitness = sum(fitness_copy_inverted)
    probabilities = [fitness / total_fitness for fitness in fitness_copy_inverted]
    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 != len(new_population) - 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).
    """
    random_indexes = np.random.permutation(self.population_size)

    for i in range(0, len(random_indexes) - 1, 2):
        # With probability.
        if random() < self.crossover_probability:
            # Generate uniform random number (0, 1)
            u = 0.
            while u == 0.:
                u = random()

            degree = 1 / (self.n + 1)

            if u > 0.5:
                beta = (1 / (2 * (1 - u))) ** degree
            else:
                beta = (2 * u) ** degree

            p1 = self.population[random_indexes[i]]
            p2 = self.population[random_indexes[i + 1]]

```

```

        c1 = 0.5 * ((1 + beta) * p1 + (1 - beta) * p2)
        c2 = 0.5 * ((1 - beta) * p1 + (1 + beta) * p2)

        self.population[random_indexes[i]] = c1
        self.population[random_indexes[i + 1]] = c2

def mutate(self, generation, left_limit, right_limit, b):
    """
    Non-uniform Mutation for population with probability
    :param generation: number of generation
    :param left_limit: left boundary of mutate value
    :param right_limit: right boundary of mutate value
    :param b: degree of an influence from generations number
    """
    for i in range(self.population_size):
        if random() < self.mutation_probability:
            # Dimension (component of chromosome) to mutate.
            rand_dimension = randint(0, self.n - 1)
            chromosome_component = self.population[i][rand_dimension]

            # Random direction to mutate.
            direction = randint(0, 1)

            if direction == 1:
                delta = count_mutation_delta(generation,
self.max_generations, right_limit, b)
                self.population[i][rand_dimension] = chromo-
some_component + delta
            else:
                delta = count_mutation_delta(generation,
self.max_generations, left_limit, b)
                self.population[i][rand_dimension] = chromo-
some_component - delta

def draw_plot(self, generation):
    dots_n = 100

    x = np.linspace(self.left_edge, self.right_edge, dots_n)
    Z = np.zeros(shape=(dots_n, dots_n))

    for i in range(dots_n):
        for j in range(dots_n):
            Z[i][j] = self.fitness_function(np.array([x[i], x[j]]))

    # Plot the surface
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    X, Y = np.meshgrid(x, x)
    ax.plot_surface(X, Y, Z, cmap='inferno', alpha=0.7)

    x_dots = self.population[:, 0]
    y_dots = self.population[:, 1]
    z_dots = self.fitness_values

    ax.scatter3D(x_dots, y_dots, z_dots, color='green', marker='o',
s=50, edgecolor='black')

    ax.set_xlabel("X-axis")
    ax.set_ylabel("Y-axis")
    ax.set_zlabel("Z-axis")
    ax.set_title(f"Generation {generation}. Population size is

```

```

{self.population_size}.\n" +
        f"Pc -> {self.crossover_probability}. Pm ->
{self.mutation_probability}. " +
        f"Min fitness = {min(self.fitness_values):.4f}")

plt.show()

import numpy as np
import random
from deap import base, creator, tools, algorithms

# Функция фитнеса (Rastrigin's function)
def fitness_function(chromosome):
    chromosome = np.array(chromosome) # Преобразуем список в numpy массив
    return 10 * len(chromosome) + np.sum(np.power(chromosome, 2) - 10 *
np.cos(2 * np.pi * chromosome)),

# Настройка генетического алгоритма с помощью DEAP
creator.create("FitnessMin", base.Fitness, weights=(-1.0,)) # Для минимизации
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()

# Параметры индивидуума (длина хромосомы и диапазон значений)
CHROMOSOME_LENGTH = 2 # Длина хромосомы (количество параметров)
BOUND_LOW, BOUND_HIGH = -5.12, 5.12 # Границы поиска

# Определим хромосому как список реальных чисел в пределах от BOUND_LOW до
BOUND_HIGH
toolbox.register("attr_float", random.uniform, BOUND_LOW, BOUND_HIGH)
toolbox.register("individual", tools.initRepeat, creator.Individual,
toolbox.attr_float, CHROMOSOME_LENGTH)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Функция оценки (фитнес)
toolbox.register("evaluate", fitness_function)

# Операторы генетического алгоритма
toolbox.register("mate", tools.cxBlend, alpha=0.5) # Оператор скрещивания
(Blend Crossover)
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=1, indpb=0.1) #
Оператор мутации (гауссовское распределение)
toolbox.register("select", tools.selTournament, tournsize=3) # Турнирная
селекция

# Параметры алгоритма
toolbox.register("map", map)

# Основной процесс ГА
def main():
    random.seed(42)

    # Инициализация популяции
    population = toolbox.population(n=100)

    # Статистика для отслеживания прогресса
    stats = tools.Statistics(lambda ind: ind.fitness.values)

```

```

stats.register("avg", np.mean)
stats.register("min", np.min)

# Запуск генетического алгоритма
population, logbook = algorithms.eaSimple(population, toolbox,
cxpb=0.5, mutpb=0.2, ngen=50,
                                         stats=stats, verbose=True)

# Найти и вывести лучший индивидиум
best_individual = tools.selBest(population, 1)[0]
print(f"Best individual is {best_individual}, with fitness:
{best_individual.fitness.values[0]}")

if __name__ == "__main__":
    main()

```

4 Вывод

В ходе работы была создана программа для решения оптимизационной задачи с использованием вещественного кодирования хромосом. С помощью генетического программирования была решена задача оптимизации многомерной функции.