МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ
И ПРОГРАММНОЙ ИНЖЕНЕРИИ (КАФЕДРА №43)

ПРЕПОДАВАТЕЛЬ

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

ОТЧЁТ О ЛАБОРАТОРНОЙ РАБОТЕ № 2

ГЕНЕТИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

по дисциплине: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР.		
	подпись, дата	инициалы, фамилия

Санкт-Петербург
2024

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.

2. Индивидуальное задание по варианту

Вариант 10

Ackley’s Path

3. Теоретические сведения

Генетический алгоритм — это эвристический алгоритм поиска, используемый для решения задач оптимизации и моделирования путём случайного подбора, комбинирования и вариации искомых параметров с использованием механизмов, аналогичных естественному отбору в природе.

Является разновидностью эволюционных вычислений, с помощью которых решаются оптимизационные задачи с использованием методов естественной эволюции, таких как мутации, отбор и кроссинговер.

В данной работе были реализованы следующие операторы:

1. Репродукция — это процесс, в котором хромосомы копируются в промежуточную популяцию для дальнейшего "размножения" согласно их значениям целевой (фитнес-) функции. При этом хромосомы с лучшими значениями целевой функции имеют большую вероятность попадания одного или более потомков в следующее поколение.

2. Кроссинговер. В данной работе использован min-max кроссинговер.

3. Мутация – данный оператор совершает инверсию случайного гена в хромосоме. Иногда данный оператор играет вторичную роль, так как его вероятность слишком мала, примерно 0.01%

4. Результаты выполнения программы

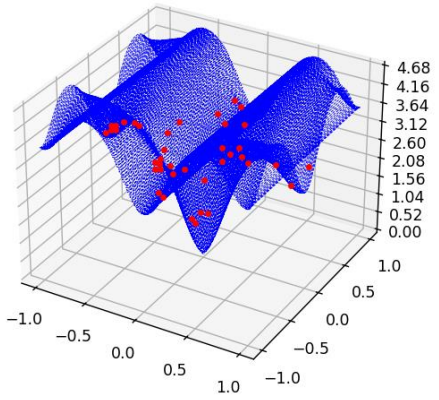
Результат выполнения при n=2

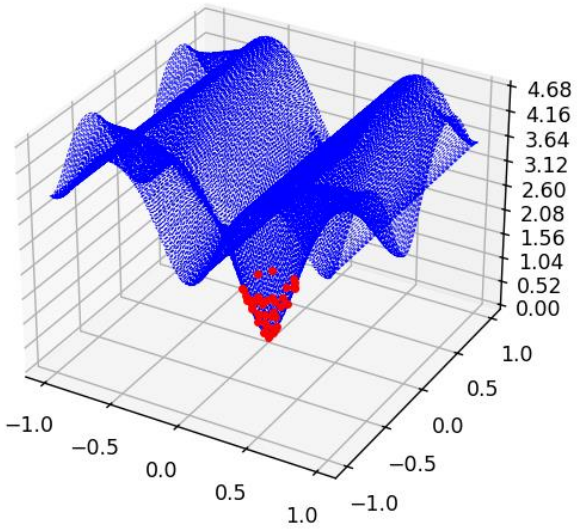
Параметры:

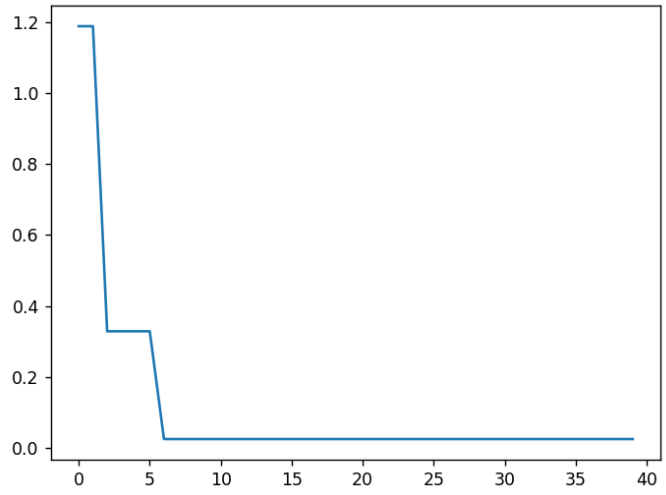
Размер популяции 40;

Вероятность кроссинговера: 0.5

Вероятность мутации: 0.001








```
solution goal (args) = [418.982, 420.968]
solution goal (val)  = 20.035824785859372
calculated goal (args) = [[None, 3.6253849384403627, 3.6253849384403627]
calculated goal (val)  = [None, 3.6253849384403627, 3.6253849384403627]
```

Результат выполнения при n=5

Параметры:

Размер популяции 40;

Вероятность кроссинговера: 0.5

Вероятность мутации: 0.001

```
population size      = 40

dispersion (goal)    = 0.001
dispersion (status)  = False

diff                 = 2.095064978923716

solution (args)      = [-0.049058073543615244, -0.19397489314527094,
-0.10953522888366884, -0.21038783043367904, -0.14703059385417006]

solution (val)       = 1.5303199595166466
calculated goal (args) = [3.6253849384403627, 3.6253849384403627,
3.6253849384403627, 3.6253849384403627, 3.6253849384403627]
calculated goal (val)  = [3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627]
```

Результат выполнения при n=5

Параметры:

Размер популяции 100;

Вероятность кроссинговера: 0.5

Вероятность мутации: 0.001

```
population size      = 100

dispersion (goal)    = 0.001
dispersion (status)  = False

diff                 = 3.2400834424570633
solution (args)      = [-0.047005433668849905, -0.06268709346668522, -0.03827166547766758, 0.09060688209458645, -0.0017714105002457625]
solution (val)       = 0.38530149598329944
time work            = 0:00:00.360201

calculated goal (args) = [3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627]
calculated goal (val)  = [ 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627]
```

Результат выполнения при n=5

Параметры:

Размер популяции 300;

Вероятность кроссинговера: 0.5

Вероятность мутации: 0.001

```
population size      = 300

dispersion (goal)    = 0.001
dispersion (status)  = False

diff                 = 3.0501454356536897
solution (args)      = [-1.1260402328350239e-06, -0.07880400094520001, -0.11038300634189846, -0.0024270852031840207, -0.09882700931076083]
solution (val)       = 0.575239502786673
time work            = 0:00:01.736290

calculated goal (args) = [ 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627]
calculated goal (val)  = [ 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627]
```

Результат выполнения при n=5

Параметры:

Размер популяции 40;

Вероятность кроссинговера: 0.9

Вероятность мутации: 0.001

```
population size      = 40

dispersion (goal)    = 0.001
dispersion (status)  = False

diff                 = 2.5073016854329815
solution (args)      = [-0.06587209183007614, 0.23027351187309186, 0.06667291210788062, -0.11019697397933226, -0.046412857752838876]
solution (val)       = 1.1180832530073812
time work            = 0:00:00.246074

calculated goal (args) = [ 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627]
calculated goal (val)  = [ 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627]
```

Результат выполнения при n=5

Параметры:

Размер популяции 40;

Вероятность кроссинговера: 0.1

Вероятность мутации: 0.001

```
population size      = 40

dispersion (goal)    = 0.001
dispersion (status)  = False

diff                 = 3.1334544514879603
solution (args)      = [0.002997219473939383, -0.058543065565390906, -0.12544656946517208, -0.02500493658957259, 0.053699348619090204]
solution (val)       = 0.4919304869524024
time work            = 0:00:00.134164

calculated goal (args) = [3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627]
calculated goal (val)  = [3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627]
```

Результат выполнения при n=5

Параметры:

Размер популяции 40;

Вероятность кроссинговера: 0.5

Вероятность мутации: 0.1

```
population size      = 40

dispersion (goal)    = 0.001
dispersion (status)  = False

diff                 = 1.9006407897649247
solution (args)      = [-0.224478097988573, -0.06872260997697421, -0.04904992609929115, 0.17597776728411696, 0.23798553296974556]
solution (val)       = 1.724744148675438
time work            = 0:00:00.190001

calculated goal (args) = [  3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627]
calculated goal (val)  = [  3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627]
```

Результат выполнения при n=5

Параметры:

Размер популяции 40;

Вероятность кроссинговера: 0.5

Вероятность мутации: 0.5

```
population size      = 40
loop                 = 39
dispersion (goal)    = 0.001
dispersion (status)  = False

diff                 = 1.9291464646292056
solution (args)      = [0.06485505374501233, 0.22891917683953888, -0.09908968576483534, 0.20809365487436415, -0.1724281953361222]
solution (val)       = 1.696238473811157
time work            = 0:00:00.267859

calculated goal (args) = [  3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627]
calculated goal (val)  = [ 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627]
```

Результат выполнения при n=10

Параметры:

Размер популяции 1000;

Вероятность кроссинговера: 0.5

Вероятность мутации: 0.5

```
population size      = 1000

dispersion (goal)    = 0.001
dispersion (status)  = False

diff                 = 2.681340186150372

solution (args)      = [0.0014027281989339446, -0.11253814083501745, -0.1265095507662497, -0.06825570586841634,
0.09400023693776327, -0.04270081638284351, -0.02044755443669044, 0.028385288468311343, -0.16050023987482187, -0.2145431077637019]

solution (val)       = 0.9440447522899906
time work            = 0:00:28.792493


calculated goal (args) = [3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627,
3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627]

calculated goal (val)  = [3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627,
3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627, 3.6253849384403627]
```

Вывод

В ходе выполнения лабораторной работы была изучена оптимизация с помощью многомерных функций ГА. Для поиска решения при n = 10 потребовалось значительно больше времени, чем для n = 5. Можем сделать вывод, что чем больше вероятность мутации, тем больше время выполнения программы. Точность полученных результатов тоже повысилась. Также чем меньше вероятность кроссиговвера тем больше увеличивается время поиска значения и точность получаемых значений.

Код программы

Params.py

```
from math import cos, exp, sqrt, pi
import numpy as np

step = 0.01

def f(*x):
    a = 20
    b = 0.2
    c = 2 * pi
    sum1 = sum(xi ** 2 for xi in x)
    sum2 = sum(cos(c * xi) for xi in x)
    term1 = -a * exp(-b * sqrt(sum1 / len(x)))
    term2 = -exp(sum2 / len(x))
    return term1 + term2 + a + exp(1)


def find_min(func, x_min, x_max, n):
    print("Start calc min")

    if not (n == 2 or n == 3 or n == 5 or n == 10):
        raise Exception("!!!")

    x = np.arange(x_min, x_max, step)

    f_min = [None] * (n + 1)
    fx = [x_min] * (n + 1)

    for i in range(1, n + 1):
        f_min[i] = func(*[x_min] * i + [x_max] * (n - i))
        fx[i] = [x_min] * i + [x_max] * (n - i)

    print("Finish calc min")

    return f_min, *fx


class Params:
    class F:
        x_min = -1
        x_max = 1
        step = 0.01
        # Uncomment one of the following lines based on your desired
dimension
        n = 2
        f_min, *fx_min = find_min(f, x_min, x_max, n)

    class Probability:
        crossing_over = 0.1
        mutation = 0.001

    class Accuracy:
        goal_epoch = 40
        dispersion = 0.001

    class Population:
        init_size = 1000

    class Drawing:
```

```
        linewidth = 0.5
        markersize = 3
        pause = pow(2, -32)

    class Log:
        probability = False
        operator_result = False

    class Debug:
        chromosomes_counter = 0

    # Добавим атрибуты fx1_min, fx2_min, ..., fx10_min
    F.fx1_min, F.fx2_min, F.fx3_min, F.fx4_min, F.fx5_min, F.fx6_min,
F.fx7_min, F.fx8_min, F.fx9_min, F.fx10_min = [None] * 10
```

Util.py

```
import random
from Params import *


def true_with_probability(probability: float):
    return random.uniform(0.0, 1) <= probability


def merge_lists(lst1: list, lst2: list) -> list:
    l1 = list(lst1)
    l2 = list(lst2)
    for i in l2:
        l1.append(i)

    return l1


def gen_x1_x2():
    x_min = Params.F.x_min
    x_max = Params.F.x_max
    step = Params.F.step
    x = np.arange(x_min, x_max, step)

    x1 = []
    for i in x:
        for _ in range(len(x)):
            x1.append(i)

    x2 = []
    for _ in range(len(x)):
        for i in x:
            x2.append(i)

    return x1, x2
```

Chromosome.py

```
import numpy as np

import Params
from Util import *
```

```
class Chromosome:
    obj_id: int
    values: list[float]

    def __init__(self, val: list[float]):
        self.obj_id = Params.Debug.chromosomes_counter
        Params.Debug.chromosomes_counter += 1
        self.values = val

    def __repr__(self):
        return "<" + "{: >3}".format(str(self.obj_id)) + "> " +
str(self.values)

    def set_values(self, val: list[float]):
        self.values = val

    def get_values(self):
        return self.values


def generate_chromosomes(count: int):
    min_val = Params.F.x_min
    max_val = Params.F.x_max
    if Params.F.n == 2:
        return [Chromosome([random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val)])
                for _ in range(count)]

    if Params.F.n == 3:
        return [Chromosome([random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val)])
                for _ in range(count)]

    if Params.F.n == 5:
        return [Chromosome([random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val)])
                for _ in range(count)]

    if Params.F.n == 10:
        return [Chromosome([random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val),
                            random.uniform(min_val, max_val)])
                for _ in range(count)]


def sort_population(population: list[Chromosome], func) -> list[Chromosome]:
    if Params.F.n == 2:
        x1, x2, y = calc_f(population, func)

        yxx = np.array([y, x1, x2]).T.tolist()
        # sorted_yxx = sorted(yxx, reverse=True)
        sorted_yxx = sorted(yxx)
```

```
        return [Chromosome([i[1], i[2]]) for i in sorted_yxx]

    if Params.F.n == 3:
        x1, x2, x3, y = calc_f(population, func)

        yxxx = np.array([y, x1, x2, x3]).T.tolist()
        # sorted_yxx = sorted(yxx, reverse=True)
        sorted_yxxx = sorted(yxxx)

        return [Chromosome([i[1], i[2], i[3]]) for i in sorted_yxxx]

    if Params.F.n == 5:
        x1, x2, x3, x4, x5, y = calc_f(population, func)

        y5 = np.array([y, x1, x2, x3, x4, x5]).T.tolist()
        # sorted_yxx = sorted(yxx, reverse=True)
        sorted_y5 = sorted(y5)

        return [Chromosome([i[1], i[2], i[3], i[4], i[5]]) for i in
sorted_y5]

    if Params.F.n == 10:
        x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, y = calc_f(population, func)

        y10 = np.array([y, x1, x2, x3, x4, x5, x6, x7, x8, x9,
x10]).T.tolist()
        # sorted_yxx = sorted(yxx, reverse=True)
        sorted_y10 = sorted(y10)

        return [Chromosome([i[1], i[2], i[3], i[4], i[5], i[6], i[7], i[8],
i[9], i[10]]) for i in sorted_y10]


# Метод: отбор на основе усечения
# Сначала отбираемые особи упорядочиваются согласно их значениям целевой
функции.
# Затем в качестве родителей выбираются только лучшие особи.
# Далее, с равной вероятностью, среди них случайным образом выбирают пары,
которые производят потомков.
# Порог отсечения T - часть популяции, которая отбирается в качестве
родителей. Обычно 10-50 %
def reproduction(population: list[Chromosome], func) -> (list[Chromosome],
list[Chromosome]):
    if Params.F.n == 2:
        x1, x2, y = calc_f(population, func)

        yxx = np.array([y, x1, x2]).T.tolist()

        avg = sum(yxx[0]) / len(yxx[0])
        index = len(list(filter(lambda arg: arg[0] < avg, yxx)))
        selection = yxx[:][:index]

        parents = [Chromosome([i[1], i[2]]) for i in selection]

    if Params.F.n == 3:
        x1, x2, x3, y = calc_f(population, func)

        yxxx = np.array([y, x1, x2, x3]).T.tolist()

        avg = sum(yxxx[0]) / len(yxxx[0])
        index = len(list(filter(lambda arg: arg[0] < avg, yxxx)))
        selection = yxxx[:][:index]
```

```
        parents = [Chromosome([i[1], i[2], i[3]]) for i in selection]

    if Params.F.n == 5:
        x1, x2, x3, x4, x5, y = calc_f(population, func)

        y5 = np.array([y, x1, x2, x3, x4, x5]).T.tolist()

        avg = sum(y5[0]) / len(y5[0])
        index = len(list(filter(lambda arg: arg[0] < avg, y5)))
        selection = y5[:][:index]

        parents = [Chromosome([i[1], i[2], i[3], i[4], i[5]]) for i in
selection]

    if Params.F.n == 10:
        x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, y = calc_f(population, func)

        y10 = np.array([y, x1, x2, x3, x4, x5, x6, x7, x8, x9,
x10]).T.tolist()
        avg = sum(y10[0]) / len(y10[0])
        index = len(list(filter(lambda arg: arg[0] < avg, y10)))
        selection = y10[:][:index]

        parents = [Chromosome([i[1], i[2], i[3], i[4], i[5], i[6], i[7],
i[8], i[9], i[10]]) for i in selection]

    return parents


def round_gen(gen: float):
    if gen <= Params.F.x_min:
        return Params.F.x_min
    elif gen >= Params.F.x_max:
        return Params.F.x_max
    else:
        return gen


def crossing_over(parents: list[Chromosome]) -> list[Chromosome]:
    new_chromosomes: list[Chromosome] = []
    for i in range(len(parents)):
        if true_with_probability(Params.Probability.crossing_over) and
len(parents) > 1:
            if Params.Log.probability:
                print("crossing_over ◉ ")
            parent_id = random.randint(0, len(parents) - 1)
            while parent_id == i:
                parent_id = random.randint(0, len(parents) - 1)

            parent1 = parents[i].get_values()
            parent2 = parents[parent_id].get_values()
            child1 = Chromosome([0 for i in
range(len(parent1))]).get_values()
            child2 = Chromosome([0 for i in
range(len(parent1))]).get_values()
            child3 = Chromosome([0 for i in
range(len(parent1))]).get_values()
            for j in range(len(parents[parent_id].values)):
                child1[j] = round_gen(0.5 * parent1[j] + 0.5 * parent2[j])
                child2[j] = round_gen(1.5 * parent1[j] - 0.5 * parent2[j])
                child3[j] = round_gen(0.5 * parent1[j] + 1.5 * parent2[j])

            new_chromosomes.append(Chromosome(child1))
            new_chromosomes.append(Chromosome(child2))
```

```
            new_chromosomes.append(Chromosome(child3))

    return new_chromosomes


# случайная мутация
def mutation(population: list[Chromosome]) -> list[Chromosome]:
    for chromosome in population:
        if true_with_probability(Params.Probability.mutation):
            if Params.Log.probability:
                print("mutation ◉      ")
            genes = list(chromosome.get_values())
            i = random.randint(0, len(genes) - 1)
            new_value = random.uniform(
                Params.F.x_min, Params.F.x_max)
            genes[i] = new_value
            chromosome.set_values(genes)

    return population


def calc_f(population, func):
    args = list(map(Chromosome.get_values, population))

    if Params.F.n == 2:
        x1, x2 = np.array(args).T.tolist()
        y = list(map(func, x1, x2))
        return x1, x2, y

    if Params.F.n == 3:
        x1, x2, x3 = np.array(args).T.tolist()
        y = list(map(func, x1, x2, x3))
        return x1, x2, x3, y

    if Params.F.n == 5:
        x1, x2, x3, x4, x5 = np.array(args).T.tolist()
        y = list(map(func, x1, x2, x3, x4, x5))
        return x1, x2, x3, x4, x5, y

    if Params.F.n == 10:
        x1, x2, x3, x4, x5, x6, x7, x8, x9, x10 = np.array(args).T.tolist()
        y = list(map(func, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10))
        return x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, y
```

Evolution.py

```
import datetime
import matplotlib.pyplot as plt
from matplotlib.ticker import LinearLocator

from Chromosome import *


class Evolution:
    generation = generate_chromosomes(Params.Population.init_size)

    def run(self):
        start_time = datetime.datetime.now()
        loop = 0
        best_in_population = []
        global_best = 0
        fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
```

```
        while loop < Params.Accuracy.goal_epoch and \
                not dispersion(self.generation, Params.Accuracy.dispersion):

            if not len(self.generation):
                self.generation =
generate_chromosomes(Params.Population.init_size)

            print("↓ ↓ ↓ ↓ ↓ ШАГ ↓ ↓ ↓ ↓ ↓")

            # print("in: " + str(self.generation))

            self.generation = sort_population(self.generation, f)
            parents = list(reproduction(self.generation, f))
            new_chromosomes = list(crossing_over(parents))
            self.generation = merge_lists(self.generation, new_chromosomes)
            self.generation = list(mutation(self.generation))
            self.generation = merge_lists(self.generation,
generate_chromosomes(Params.Population.init_size))
            self.generation = sort_population(self.generation, f)
            # cut generation
            self.generation = self.generation[:Params.Population.init_size]

            # print("out: " + str(self.generation))

            if Params.F.n == 2:
                x11, x12 = gen_x1_x2()
                y1 = list(map(f, x11, x12))
                x21, x22, y2 = calc_f(self.generation, f)

                ax.cla()
                ax.zaxis.set_major_locator(LinearLocator(10))
                ax.zaxis.set_major_formatter('{x:.02f}')
                ax.plot(x11, x12, y1, '--b',
linewidth=Params.Drawing.linewidth)
                ax.plot(x21, x22, y2, 'ro',
markersize=Params.Drawing.markersize)
                plt.draw()
                plt.pause(Params.Drawing.pause)

            print("population size      = " + str(len(self.generation)))
            print("loop                 = " + str(loop))
            print("dispersion (goal)    = " +
str(Params.Accuracy.dispersion))
            print("dispersion (status)  = " + str(dispersion(self.generation,
Params.Accuracy.dispersion)))
            print("Params.F.f_min type =", type(Params.F.f_min))
            print("Function value type =",
type(f(*self.generation[0].get_values())))
            print("diff                 =", Params.F.f_min[Params.F.n] -
f(*self.generation[0].get_values()))
            print("solution (args)      = " +
str(self.generation[0].get_values()))
            print("solution (val)       = " +
str(f(*self.generation[0].get_values())))
            print("time work            = " + str(datetime.datetime.now() -
start_time))
            print("↑ ↑ ↑ ↑ ↑ Шаг ↑ ↑ ↑ ↑ ↑")
            print("-----------------------------------------")
            print()

            best_in_population.append(f(*self.generation[0].get_values()))
            loop += 1
```

```
        print("-----------------------------------------")
        if Params.F.n == 2:
            goal_solution = [418.982, 420.968]
            print("solution goal (args) = " + str(goal_solution))
            print("solution goal (val)  = " + str(f(*goal_solution)))

        print("calculated goal (args) = " + str([Params.F.f_min,
Params.F.fx1_min, Params.F.fx2_min, Params.F.fx3_min, Params.F.fx4_min,
Params.F.fx5_min, Params.F.fx6_min, Params.F.fx7_min, Params.F.fx8_min,
Params.F.fx9_min, Params.F.fx10_min]))
        print("calculated goal (val)  = " + str(Params.F.f_min))

        print("-----------------------------------------")
        print("★★★★☆☆☆☆☆☆☆☆ END ☆☆☆☆☆☆☆☆★★★★")
        print("-----------------------------------------")

        if Params.F.n == 2:
            plt.show()

        fig1, ax1 = plt.subplots(subplot_kw={"projection": "rectilinear"})
        ax1.cla()
        ax1.plot(range(len(best_in_population)), best_in_population)
        plt.draw()
        plt.show()


def dispersion(population: list[Chromosome], epsilon: float) -> bool:
    summary = 0
    count = 0
    for chromosome in population:
        summary += abs(sum(chromosome.get_values()))
        count += 1

    avg = summary / count
    for chromosome in population:
        if abs(sum(chromosome.get_values()) - avg) > epsilon:
            return False
    return True
```

Main.py

```
from Evolution import *

# Создать экземпляр Evolution и запустить эволюцию
Evolution().run()
```