ГУАП

КАФЕДРА № 43

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

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

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №2 |
| Генетический алгоритм вещественной оптимизации |
| по курсу: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

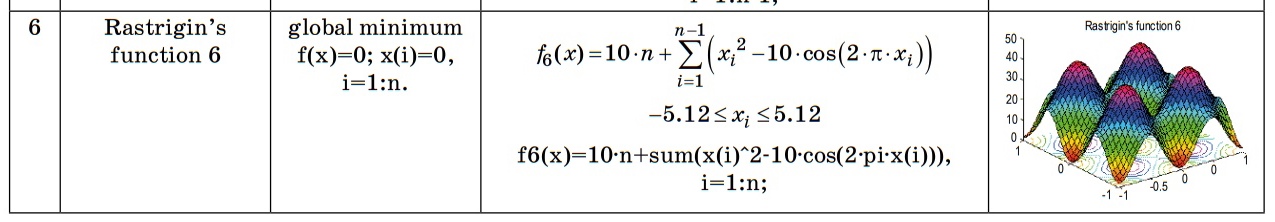
РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4136 |  |  |  | Якшин С.Е. |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2024

**Индивидуальное задание по варианту:**

Вариант 22 (6)



**Цель работы:**

модификация представления хромосомы и операторов рекомбинации ГА для оптимизации многомерных функций.

Графическое отображение результатов оптимизации.

**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 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 алгоритмов с вещественным кодированием:

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 real-coded алгоритмов.

21

Также рекомендуется использовать стратегию элитизма – лучшая особь сохраняется отдельно и не стирается при смене эпох, принимая при этом участие в отборе и рекомбинации.

**Код программы:**

#ifndef LAB2

#define LAB2

#include <iostream>

#include <vector>

#include <cmath>

#include <functional>

#include <cstdint>

#include <random>

#include "Plot.h"

#include "Bocchiemark.hpp"

namespace lab2

{

constexpr uint8\_t n = 2;

typedef std::array<float, n> Entity;

typedef std::vector<Entity> Population;

typedef std::vector<float> Fitness;

namespace random

{

int \_int(int l, int r)

{

static std::random\_device rd;

static std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dist(l, r);

return dist(gen);

}

float \_float(float l, float r)

{

static std::random\_device rd;

static std::mt19937 gen(rd());

std::uniform\_real\_distribution<> dist(l, r);

return (float) dist(gen);

}

}

namespace k

{

const size\_t EPOCH\_CRITERIA = 300;

const size\_t EPOCH\_LIMIT = 200'000;

const size\_t FLOAT\_EQ\_PRECISION = 5;

const float MIN = -5.12, MAX = 5.12;

struct P

{

float \_CROSSING = 0.5; // Вероятность скрещивания

float \_MUTATION = 0.0001; // Вероятность мутации

};

uint16\_t N = 100;

inline float f(const Entity &x)

{

float result = 10.0 \* x.size();

for (float xi : x) {

result += xi \* xi - 10.0 \* std::cos(2.0 \* M\_PI \* xi);

}

return result;

}

inline std::vector<float> f(const Population &x)

{

std::vector<float> \_;

for (const auto &\_\_ : x)

\_.push\_back(f(\_\_));

return \_;

}

// Функция для сравнения двух чисел с плавающей запятой с точностью до n знаков

bool are\_equal\_with\_precision(float a, float b, int precision = FLOAT\_EQ\_PRECISION) {

// Степень 10 для проверки точности

float epsilon = std::pow(10.0f, -precision);

// Проверяем разницу между числами

return std::fabs(a - b) < epsilon;

}

}

namespace op

{

Population selection(const Population &population,

const Fitness &fitness,

int num\_selected)

{

std::vector<std::pair<float /\* <- fit \*/ , uint16\_t /\* <- source ind \*/>> sorted\_fitness;

for (size\_t i = 0; i < fitness.size(); ++i)

sorted\_fitness.emplace\_back(fitness[i], i);

std::sort(sorted\_fitness.begin(), sorted\_fitness.end(), [](const auto &a, const auto &b)

{

return a.first < b.first;

});

Population selected\_population;

for (int i = 0; i < num\_selected; ++i) {

selected\_population.push\_back(population[sorted\_fitness[i].second]);

}

return selected\_population;

}

Population crossover(const Population &population) // SBX crossover

{

Population new\_population;

for (size\_t i = 0; i < size\_t(population.size() / 2); ++i) {

const auto &parent1 = population[i];

const auto &parent2 = population[2 \* i];

Entity child1, child2;

for (size\_t j = 0; j < n; ++j) {

// Рассчитываем параметр beta, который зависит от случайной величины u

float u = random::\_float(0.f, 1.f);

float beta;

if (u <= 0.5f) {

float beta\_q = std::pow(2 \* u, 1.0f / (n + 1));

beta = beta\_q;

}

else {

float beta\_q = std::pow(1 / (2 \* (1 - u)), 1.0f / (n + 1));

beta = beta\_q;

}

float c1 = parent1[j];

float c2 = parent2[j];

child1[j] = 0.5f \* ((1 + beta) \* c1 + (1 - beta) \* c2);

child2[j] = 0.5f \* ((1 - beta) \* c1 + (1 + beta) \* c2);

}

new\_population.push\_back(child1);

new\_population.push\_back(child2);

}

return new\_population;

}

Population mutate(const Population &population, float mutation\_rate) // random mutation

{

Population mutated\_population = population;

for (Entity &entity : mutated\_population) {

if (random::\_float(0.f, 1.f) < mutation\_rate) {

decltype(n) lucker = random::\_int(0, n - 1);

float mutation = random::\_float(k::MIN, k::MAX);

entity[lucker] = mutation;

}

}

return mutated\_population;

}

}

// Генерация данных для построения графика

inline void generateData(std::vector<std::vector<float>> &x\_list, std::vector<float> &y)

{

float step = 0.1; // Шаг между точками

std::vector<float> x1;

std::vector<float> x2;

for (float i = k::MIN; i <= k::MAX; i += step) {

x1.push\_back(i);

x2.push\_back(i);

}

for (float xi : x1) {

for (float xj : x2) {

x\_list[0].push\_back(xi);

x\_list[1].push\_back(xj);

y.push\_back(k::f({xi, xj}));

}

}

}

struct aboba

{

Entity x;

float y;

std::string to\_string()

{

std::string out;

for (size\_t i = 0; i < n; ++i)

out += "x" + std::to\_string(i) + " = " + std::to\_string(x[i]) + ", ";

out += "y = " + std::to\_string(y);

return out;

}

};

Population genRandPopulation()

{

Population population;

population.reserve(k::N);

for (size\_t i = 0; i < k::N; ++i) {

Entity \_;

for (size\_t j = 0; j < n; ++j)

\_[j] = random::\_float(k::MIN, k::MAX);

population.push\_back(\_);

}

return population;

}

std::pair<int, aboba> \_main(k::P p, Population population, bool show\_plots, std::string bmark\_name = "")

{

Population source\_population = population;

std::vector<std::vector<float>> x\_list(n);

std::vector<float> y;

generateData(x\_list, y);

if (show\_plots) {

std::vector<std::vector<float>> p\_list;

p\_list.reserve(population.size());

for (const auto &\_p : population) {

p\_list.emplace\_back();

for (const auto &gene : \_p)

p\_list.back().push\_back(gene);

}

std::vector<float> p;

for (const auto &\_p : population)

p.push\_back(k::f(\_p));

plot::print3d(x\_list, y, p\_list, p);

}

bool exit = false;

size\_t epoch\_counter = 0;

auto min = [](const Population& population)

{

return \*std::min\_element(population.begin(), population.end(),

[](const auto &a, const auto &b)

{

return k::f(a) < k::f(b);

});

};

auto fine\_optimum = [&epoch\_counter, &min](const Population &population) -> bool

{

static size\_t best\_optimum\_counter = 0;

static float last\_optimum = 0xFFFFFF;

float optimum = k::f(min(population));

if (not k::are\_equal\_with\_precision(last\_optimum, optimum))

{

last\_optimum = optimum;

best\_optimum\_counter = 0;

}

else best\_optimum\_counter++;

return best\_optimum\_counter == k::EPOCH\_CRITERIA || epoch\_counter == k::EPOCH\_LIMIT;

};

// for (int i = 0; i < 50; ++i)

while (not exit) {

epoch\_counter++;

// 1. Выбор родителей для процесса размножения (работает оператор селекции - репродукции)

Population parents = op::selection(population, k::f(population), k::N \* p.\_CROSSING);

// 2. Создание потомков выбранных пар родителей (работает оператор скрещивания - кроссинговера)

Population childs = op::crossover(parents);

// 3. Мутация новых особей (работает оператор мутации)

childs = op::mutate(childs, p.\_MUTATION);

// 4. Расширение популяции за счет добавления новых только что порожденных особей

population.insert(population.end(), childs.begin(), childs.end());

// 5. Сокращение расширенной популяции до исходного размера (работает оператор редукции)

population = op::selection(population, k::f(population), k::N);

exit = fine\_optimum(population);

}

// if (epoch\_counter == k::EPOCH\_LIMIT)

// std::cout << "Educated by " << epoch\_counter << " epochs ( limit )\n";

// else

// std::cout << "Educated by " << epoch\_counter - k::EPOCH\_CRITERIA << " epochs ( without EPOCH\_CRITERIA = " << k::EPOCH\_CRITERIA << " )\n";

if (show\_plots) {

std::vector<std::vector<float>> p\_list;

p\_list.reserve(population.size());

for (const auto &\_p : population) {

p\_list.emplace\_back();

for (const auto &gene : \_p)

p\_list.back().push\_back(gene);

}

std::vector<float> p;

for (const auto &\_p : population)

p.push\_back(k::f(\_p));

plot::print3d(x\_list, y, p\_list, p);

}

return {(int) epoch\_counter, aboba { min(population), k::f(min(population)) } };

}

namespace stats {

// Функция для нахождения минимального значения

int findMin(const std::vector<int>& vec) {

return \*std::min\_element(vec.begin(), vec.end());

}

aboba findMin(const std::vector<aboba>& vec) {

return \*std::min\_element(vec.begin(), vec.end(), [](const auto& a, const auto& b)

{

return a.y < b.y;

});

}

// Функция для нахождения максимального значения

int findMax(const std::vector<int>& vec) {

return \*std::max\_element(vec.begin(), vec.end());

}

// Функция для нахождения среднего значения

double findMean(const std::vector<int>& vec) {

double sum = std::accumulate(vec.begin(), vec.end(), 0);

return sum / vec.size();

}

double findMean(const std::vector<float>& vec) {

double sum = std::accumulate(vec.begin(), vec.end(), 0.0f);

return sum / vec.size();

}

// Функция для нахождения медианы

double findMedian(const std::vector<int>& vec) {

auto copy = vec;

size\_t size = copy.size();

std::sort(copy.begin(), copy.end());

if (size % 2 == 0) {

return (copy[size / 2 - 1] + copy[size / 2]) / 2.0;

} else {

return copy[size / 2];

}

}

aboba findMedian(const std::vector<aboba>& vec) {

auto copy = vec;

size\_t size = copy.size();

std::sort(copy.begin(), copy.end(), [](const auto& a, const auto& b)

{

return a.y < b.y;

});

return copy[size / 2];

}

}

inline int main()

{

size\_t runs = 100;

std::vector<Population> populations;

for (size\_t i = 0; i < runs; ++i)

populations.push\_back(genRandPopulation());

constexpr bocchie::accuracy mark\_accuracy = bocchie::accuracy::milliseconds;

bocchie::mark \_1("1"), \_2("2"), \_3("3"), \_4("4");

auto bocchiemark = [&](float crossing\_prob, float mutation\_prob, const bocchie::mark \_mark){

std::shared\_ptr<bocchie::mark> mark = std::make\_shared<bocchie::mark>(\_mark);

std::cout << "-----------------------------------------------\n"

<< "Start education with Crossing prob = " << crossing\_prob << " and Mutation prob = " << mutation\_prob << " ...\n";

std::vector<int> epochs; epochs.reserve(runs);

std::vector<aboba> optimums; optimums.reserve(runs);

for (size\_t i = 0; i < runs; ++i)

{

auto [epoch, aboba] = mark->run(\_main, k::P{crossing\_prob, mutation\_prob}, populations[i], false, mark->get\_runnable\_name());

epochs.push\_back(epoch);

optimums.push\_back(aboba);

}

std::cout << "Min epochs: " << stats::findMin(epochs) << std::endl;

std::cout << "Max epochs: " << stats::findMax(epochs) << std::endl;

std::cout << "Avg epochs: " << stats::findMean(epochs) << std::endl;

std::cout << "Median epochs: " << stats::findMedian(epochs) << std::endl;

std::cout << "MIN OPTIMUM: " << stats::findMin(optimums).to\_string() << std::endl;

std::cout << "MEDIAN OPTIMUM: " << stats::findMedian(optimums).to\_string() << std::endl;

std::cout << mark->to\_json<mark\_accuracy>() << std::endl;

};

bocchiemark(0.5, 0.0001, \_1);

// bocchiemark(0.2, 0.0001, \_2);

bocchiemark(0.7, 0.0001, \_3);

bocchiemark(0.5, 0.001, \_4);

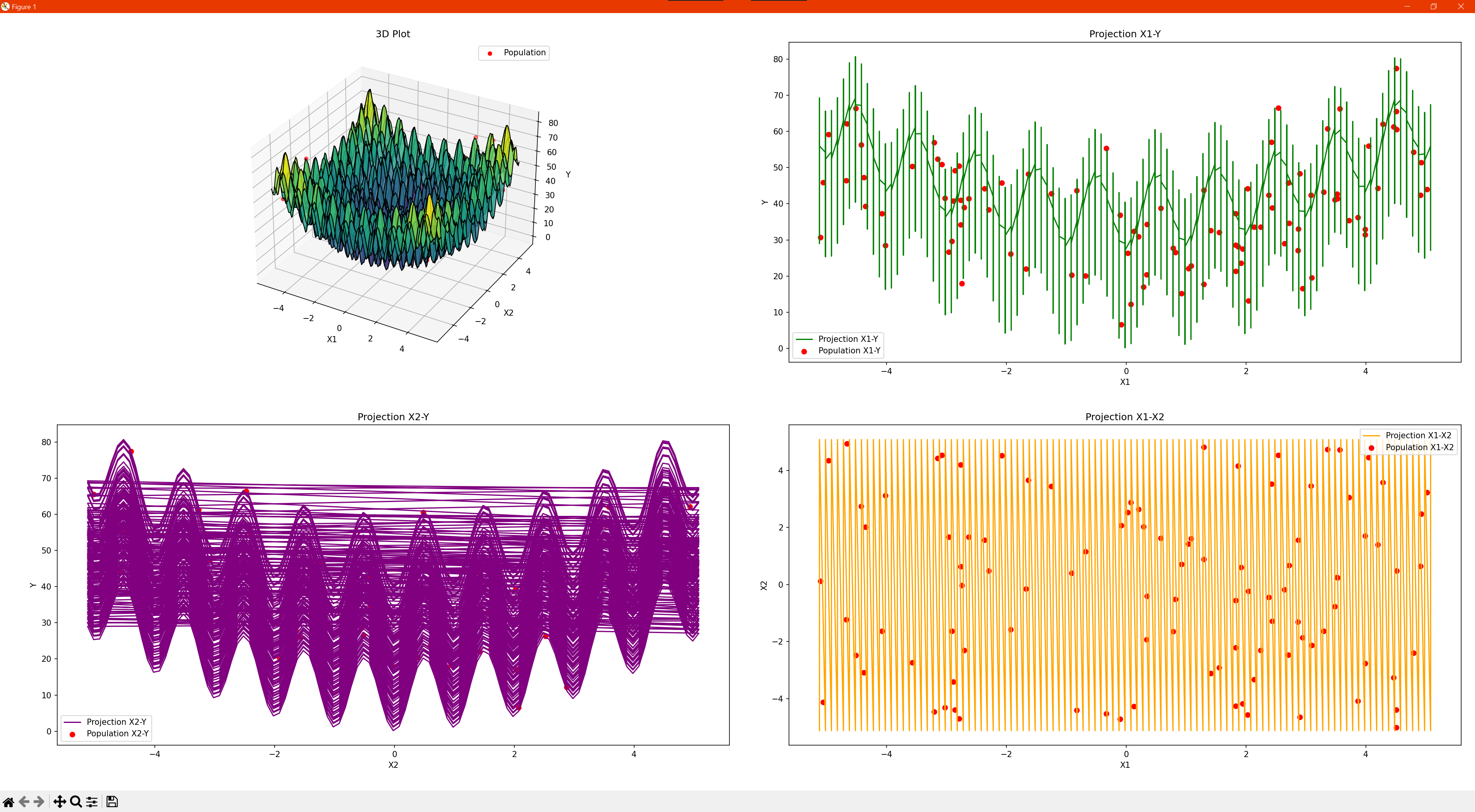
return 0;

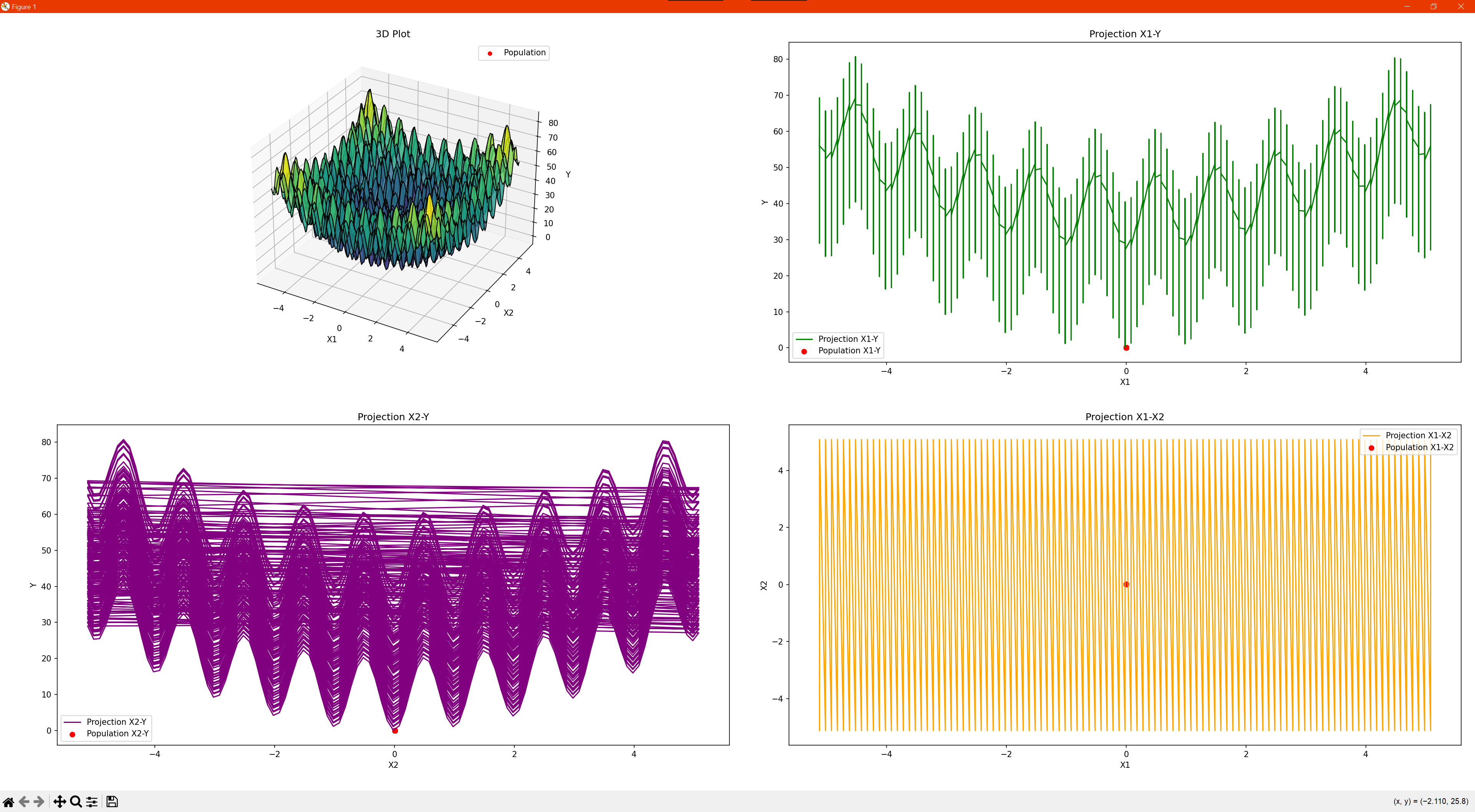
}

}

#endif

**Результаты выполнения:**

Начальное положение особей

  
Положение в конце работы алгоритма

**Экспериментальные изменения:**

Были проведены тесты обечения с различными вероятностями скрещивания и мутации.

Среди прочего, стоит отметить другие параметры, которые оставались неизменны, но существенно влияли на скорость обучения и выполнения условия «повторения лучшего результата N раз»:

1) n: 2

2) Точность сравнения особей: 5 знаков

3) Условие выхода: 300 повторений лучшего варианта

Было проведено по 100 запусков для каждого сценария, 100 наборов популяций были сгенерированы заранее, и использовались для каждого сценария:

-----------------------------------------------

Start education with Crossing prob = 0.5 and Mutation prob = 0.0001 ...

Min epochs: 307

Max epochs: 34769

Avg epochs: 3569.4

Median epochs: 447

MIN OPTIMUM: x0 = 0.000004, x1 = -0.000000, y = 0.000000

MEDIAN OPTIMUM: x0 = 0.019005, x1 = -0.000206, y = 0.071581

{

"runnable\_name": "1",

"min\_time\_milliseconds": 34.4869,

"max\_time\_milliseconds": 3806.09,

"total\_time\_milliseconds": 39308.3,

"total\_runs": 100,

"avg\_time\_milliseconds": 393.083

}

-----------------------------------------------

Start education with Crossing prob = 0.7 and Mutation prob = 0.0001 ...

Min epochs: 312

Max epochs: 21328

Avg epochs: 1530.78

Median epochs: 389.5

MIN OPTIMUM: x0 = -0.000027, x1 = -0.000000, y = 0.000000

MEDIAN OPTIMUM: x0 = -0.994963, x1 = 0.000000, y = 0.994959

{

"runnable\_name": "3",

"min\_time\_milliseconds": 39.7908,

"max\_time\_milliseconds": 2688.21,

"total\_time\_milliseconds": 19289.9,

"total\_runs": 100,

"avg\_time\_milliseconds": 192.899

}

-----------------------------------------------

Start education with Crossing prob = 0.5 and Mutation prob = 0.001 ...

Min epochs: 301

Max epochs: 17531

Avg epochs: 1323.78

Median epochs: 440

MIN OPTIMUM: x0 = 0.000038, x1 = 0.000000, y = 0.000000

MEDIAN OPTIMUM: x0 = -0.000020, x1 = 0.018797, y = 0.070014

{

"runnable\_name": "4",

"min\_time\_milliseconds": 34.1291,

"max\_time\_milliseconds": 1902,

"total\_time\_milliseconds": 14706.8,

"total\_runs": 100,

"avg\_time\_milliseconds": 147.068

}

Тут явно видно что с низким шансом скрещивания и мутации обучение идет очень долго,

Тем не менее все 3 набора не гарантируют постоянную сходимость (это видно по медианному значению)

Проведем тот же тест при n = 3:

-----------------------------------------------

Start education with Crossing prob = 0.5 and Mutation prob = 0.0001 ...

Min epochs: 335

Max epochs: 150173

Avg epochs: 25614.5

Median epochs: 14954.5

MIN OPTIMUM: x0 = -0.000114, x1 = 0.000038, x2 = -0.000024, y = 0.000002

MEDIAN OPTIMUM: x0 = -0.998589, x1 = 0.003878, x2 = -0.007055, y = 1.010429

{

"runnable\_name": "1",

"min\_time\_milliseconds": 45.1167,

"max\_time\_milliseconds": 19362.5,

"total\_time\_milliseconds": 333767,

"total\_runs": 100,

"avg\_time\_milliseconds": 3337.67

}

-----------------------------------------------

Start education with Crossing prob = 0.7 and Mutation prob = 0.0001 ...

Min epochs: 366

Max epochs: 200000

Avg epochs: 12562.3

Median epochs: 5224

MIN OPTIMUM: x0 = -0.000008, x1 = 0.000049, x2 = 0.000000, y = 0.000000

MEDIAN OPTIMUM: x0 = 0.995488, x1 = 0.004868, x2 = -0.010794, y = 1.022821

{

"runnable\_name": "3",

"min\_time\_milliseconds": 56.7163,

"max\_time\_milliseconds": 26782.5,

"total\_time\_milliseconds": 183969,

"total\_runs": 100,

"avg\_time\_milliseconds": 1839.69

}

-----------------------------------------------

Start education with Crossing prob = 0.5 and Mutation prob = 0.001 ...

Min epochs: 364

Max epochs: 13462

Avg epochs: 3328.12

Median epochs: 2769.5

MIN OPTIMUM: x0 = 0.000048, x1 = 0.000092, x2 = 0.000000, y = 0.000002

MEDIAN OPTIMUM: x0 = 0.061223, x1 = -0.000407, x2 = 0.000041, y = 0.734583

{

"runnable\_name": "4",

"min\_time\_milliseconds": 49.9666,

"max\_time\_milliseconds": 1778.93,

"total\_time\_milliseconds": 43771.9,

"total\_runs": 100,

"avg\_time\_milliseconds": 437.719

}

В целом время обучения сильно увеличилось для любого сценария

**Ответ на контрольный вопрос:**

**10.** **Каким образом определяется эффективность генетического**

**алгоритма?**

Эффективность генетического алгоритма определяется скоростью его схождение (количеством шагов) и быстротой работы алгоритма (время выполнения).