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 №5 |
| Оптимизация многомерных функций с помощью  эволюционной стратегии |
| по курсу: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

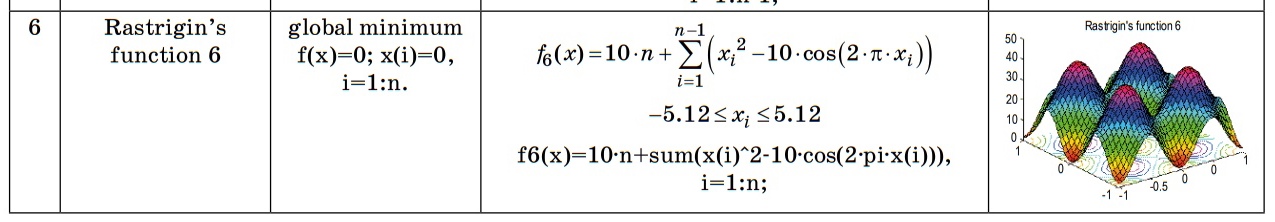
РАБОТУ ВЫПОЛНИЛ

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

Санкт-Петербург 2024

**Индивидуальное задание по варианту:**

Вариант 22 (6)



**Цель работы:**

Оптимизация функций многих переменных модификация методом эволюционной стратегии. Графическое отображение результатов оптимизации.

**Теоретические сведения:**

Эволюционные стратегии (ЭС), также, как и предыдущие парадигмы, основаны на эволюции популяции потенциальных решений, но, в отличие от них, здесь используется генетические операторы на уровне фенотипа, а не генотипа, как это делается в ГА. Разница в том, что ГА работают в пространстве генотипа – кодов решений, в то время как ЭС производят поиск в пространстве фенотипа – векторном пространстве вещественных чисел. В ЭС учитываются свойства хромосомы «в целом», в отличие от ГА, где при поиске решений исследуются отдельные гены. В природе один ген может одновременно влиять на несколько свойств организма. С другой стороны, одно свойство особи может определяться несколькими генами. Естественная эволюция основана на исследовании совокупности генов, а не отдельного (изолированного) гена.

В эволюционных стратегиях целью является движение особей популяции по направлению к лучшей области ландшафта фитнесс-функции. ЭС изначально разработаны для решения многомерных оптимизационных задач, где пространство поиска – многомерное пространство вещественных чисел. Иногда при решении задачи накладываются некоторые ограничения, например, вида gi(x)>0.

**Код программы:**

#ifndef LAB5\_HPP

#define LAB5\_HPP

#include <iostream>

#include <vector>

#include <cmath>

#include <functional>

#include <cstdint>

#include "Plot.h"

#include "Random.hpp"

#include "Bocchiemark.hpp"

namespace lab5

{

constexpr uint8\_t n = 2;

struct Entity

{

std::array<float, n> values;

std::array<float, n> biases;

};

typedef std::vector<Entity> Population;

typedef std::vector<float> Fitness;

namespace k

{

const size\_t EPOCH\_CRITERIA = 300;

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

const size\_t FLOAT\_EQ\_PRECISION = 10;

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.values.size();

for (float xi : x.values) {

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;

}

void applyReshenbergRule(Population& population, size\_t success, size\_t total) {

double successRatio = static\_cast<double>(success) / total;

// Если доля успеха больше 1/5

if (successRatio > 1.0 / 5.0) {

for (auto& individual : population) {

for (auto& param : individual.biases) {

param /= 0.82; // уменьшение на 18%

}

}

}

// Если доля успеха меньше 1/5

else if (successRatio < 1.0 / 5.0) {

for (auto& individual : population) {

for (auto& param : individual.biases) {

param \*= 0.82; // увеличение на 18%

}

}

}

}

}

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 &parents)

{

Population children;

// Размер родителей

size\_t n = parents[0].values.size();

// Для каждой пары родителей создаем одного потомка

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

Entity child1, child2;

const Entity &parent1 = parents[i];

const Entity &parent2 = parents[i + 1];

// Применяем однородный кроссовер для каждого гена

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

if (random::\_float(0.f, 1.f) < 0.5) {

child1.values[j] = parent1.values[j];

child2.values[j] = parent2.values[j];

} else {

child1.values[j] = parent2.values[j];

child2.values[j] = parent1.values[j];

}

}

children.push\_back(child1);

children.push\_back(child2);

}

return children;

}

Population mutate(const Population &population, float mutation\_rate, size\_t &successful\_mutations)

{

Population mutated\_population = population;

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

Entity &entity = mutated\_population[i];

const Entity &original\_entity = population[i];

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

// Сохраняем исходное значение для отката, если мутация не успешна

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

float original\_value = entity.values[lucker];

// Генерируем мутацию

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

entity.values[lucker] = mutation;

// Проверяем успешность мутации

if (k::f(entity) < k::f(original\_entity)) {

++successful\_mutations;

} else {

// Откатываем мутацию, если она не успешна

entity.values[lucker] = original\_value;

}

}

}

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.values[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)

{

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

\_.biases[j] = random::\_float(0, 1);

}

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.values)

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)

size\_t successful\_mutations = 0;

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, successful\_mutations);

// 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);

k::applyReshenbergRule(population, successful\_mutations, epoch\_counter);

exit = fine\_optimum(population);

}

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.values)

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 = 1;

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], true, 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(0.2, 0.1, \_4);

return 0;

}

}

#endif //LAB5\_HPP

**Результаты выполнения:**

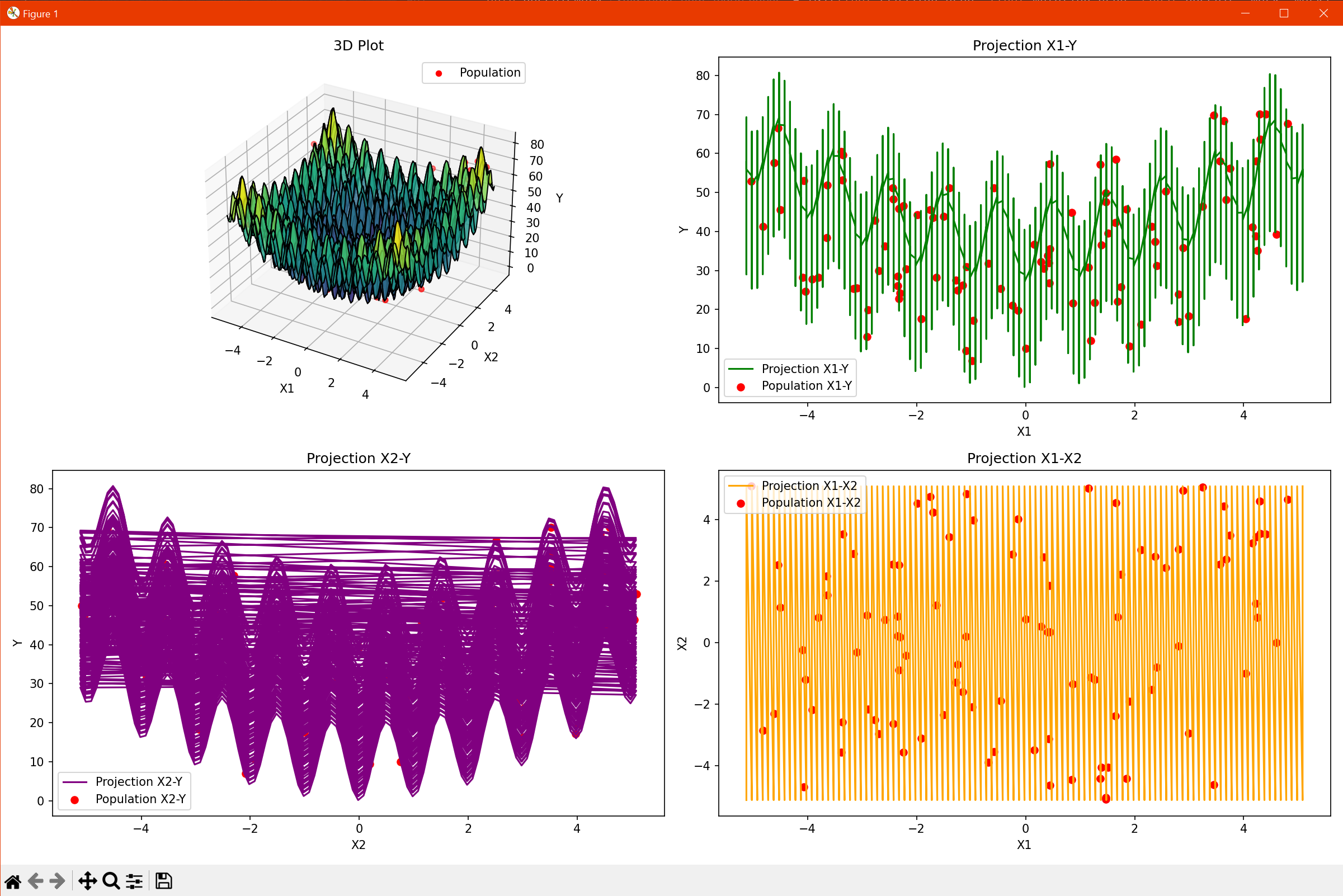


Рис. 1 – Начальная генерация особей (для функции 2-ух переменных)

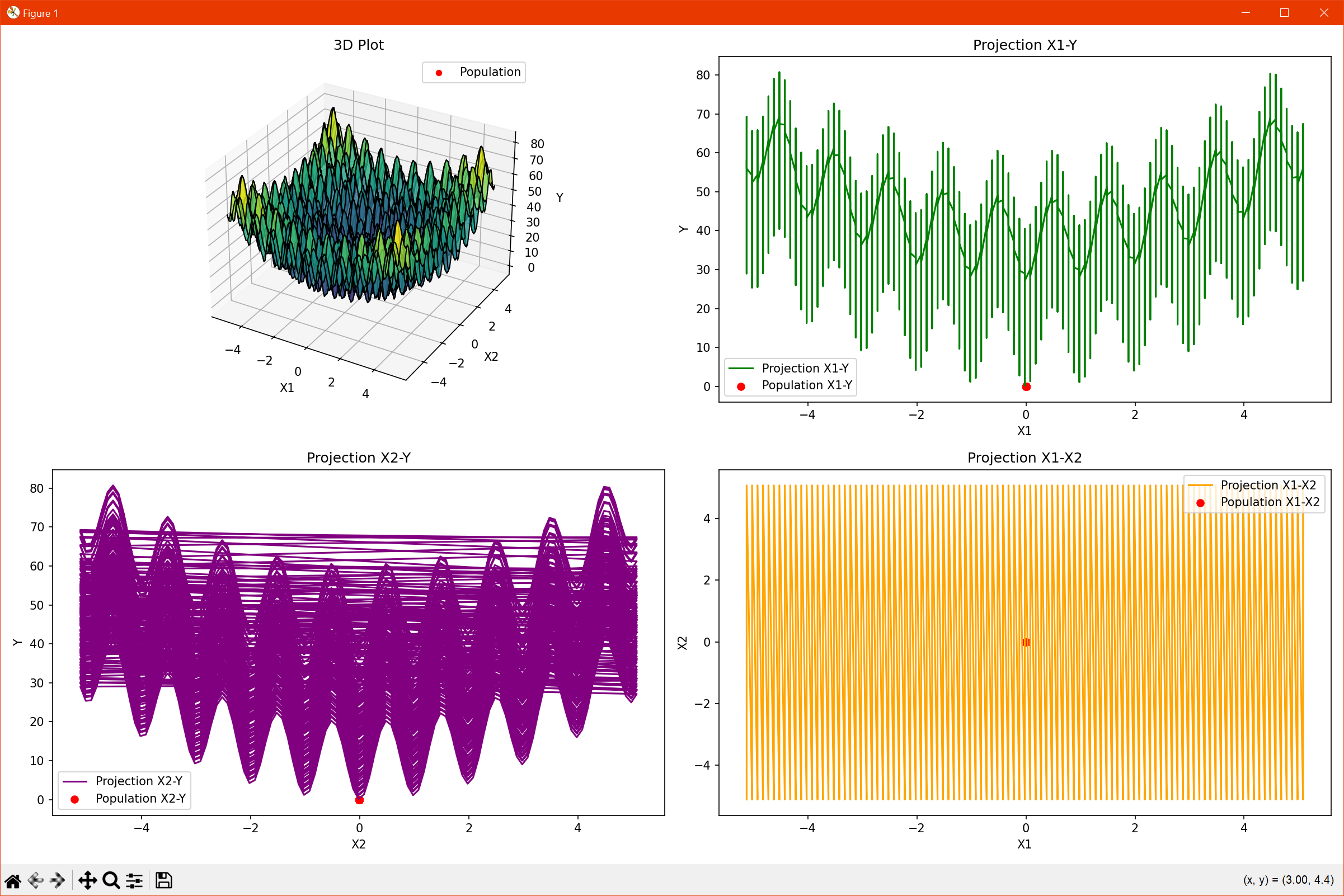


Рис. 2 – Особи для функции 2-ух переменных на момент конца работы программы

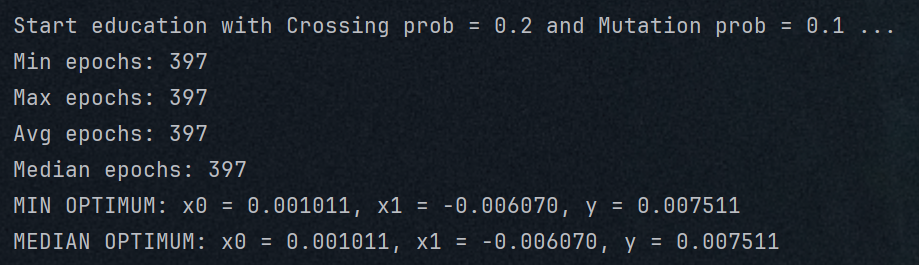


Рис. 3 – Количество эпох, за которые сошлись функции и лучшие особи для них  
  
Увеличение количества особей в популяции влечет за собой увеличение разнообразия, что помогает быстрее добиться сходимости, в купе с более высоким шансом мутации

**Ответ на контрольный вопрос:**

**10.** **Какие параметры, кроме отклонений, можно использовать в ЭС?**

В эволюционных стратегиях (ЭС) помимо отклонений можно использовать параметры, такие как размер популяции, который регулирует соотношение родителей и потомков, и тип селекции, например, элитарную или стохастическую. Также важны критерии остановки, например, по числу поколений или стабильности популяции. Влияние на мутацию оказывают параметры управления, такие как скорость изменения масштаба или использование матрицы ковариации для учета взаимозависимости между переменными. Самоадаптация параметров позволяет эволюционировать сигм вместе с генами, что улучшает гибкость алгоритма.