МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

АЭРОКОСМИЧЕСКОГО ПРИБО6РОСТРОЕНИЯ»

КАФЕДРА № 43

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Старший преподаватель |  |  |  | С.А. Рогачев |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ |
| «Оптимизация многомерных функций с помощью ГА» |
| по курсу: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4136 |  | 13.09.24 |  | А.С. Вилюмсон |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2024

**1. Задание по варианту**

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.

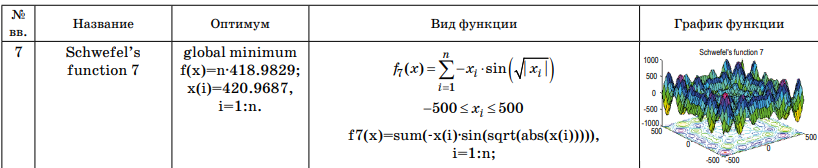
2. Для n=2 вывести на экран график данной функции с указанием найденного экстремума, точек популяции. Для вывода графиков использовать стандартные возможности пакета Matlab. Предусмотреть возможность пошагового просмотра процесса поиска решения.

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.

Вариант 7



**2. Основные сведения из теории**

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.

Основные преимущества 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.

**3. Листинг программы**

#pragma once

#include <vector>

#include <bitset>

#include <algorithm>

#include <execution>

#include <functional>

#include <random>

#include <utility>

*namespace* gen{

    static std::random\_device rd;

    static std::mt19937 gen(rd());

*double* random\_double(*double* *a*, *double* *b*) {

        std::uniform\_real\_distribution<> dis(*a*, *b*);

        return dis(gen);

    }

*long* random\_long(*long* *min*, *long* *max*){

        std::uniform\_int\_distribution<*long*> distrib(*min*, *max*);

        return distrib(gen);

    }

*namespace* reproduction{

*enum*  EXTREMUM{

            MAX = 1,

            MIN = -1

        };

*class* roulette{

*public:*

            EXTREMUM dir;

            roulette(EXTREMUM *dir* = EXTREMUM::MAX):dir(*dir*){};

            std::pair<size\_t,size\_t> doit(const std::vector<*double*>& *fitness*){

*double* sum = 0;

*double* max = \*std::max\_element(*fitness*.begin(),*fitness*.end());

*double* min = \*std::min\_element(*fitness*.begin(),*fitness*.end());

                for (*double* v : *fitness*) {

                    sum += dir == EXTREMUM::MAX ? v - min : max - v;

                }

*double* r = random\_double(0.0, sum);

*double* current\_sum = 0;

                size\_t idx1 = 0, idx2 = 0;

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

                    current\_sum += dir == EXTREMUM::MAX ? *fitness*[i] - min : max - *fitness*[i];

                    if (r < current\_sum) {

                        idx1 = i;

                        break;

                    }

                }

                r = random\_double(0.0, sum);

                current\_sum = 0;

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

                    current\_sum += dir == EXTREMUM::MAX ? *fitness*[i] - min : max - *fitness*[i];

                    if (r < current\_sum) {

                        idx2 = i;

                        break;

                    }

                }

                return {idx1,idx2};

            }

        };

*class* tournament{

*public:*

            size\_t tournament\_size;

            EXTREMUM dir;

            tournament(size\_t *ts* = 3,EXTREMUM *dir* = EXTREMUM::MAX):tournament\_size(*ts*),dir(*dir*){};

            std::pair<size\_t,size\_t> doit(const std::vector<*double*>& *fitness*){

                std::vector<size\_t> candidates;

                size\_t size = *fitness*.size();

                for (size\_t i = 0; i < tournament\_size; i++) {

                    candidates.push\_back(random\_long(0, size - 1));

                }

*auto* lambda  = [&*fitness*, this](size\_t *a*, size\_t *b*) {

                    return this->dir == EXTREMUM::MIN ? *fitness*[*a*] > *fitness*[*b*] : *fitness*[*a*] < *fitness*[*b*];

                };

                size\_t parent1\_idx = \*std::max\_element(candidates.begin(), candidates.end(),

                    lambda);

                candidates.erase(std::remove(candidates.begin(), candidates.end(), parent1\_idx), candidates.end());

                size\_t parent2\_idx = \*std::max\_element(candidates.begin(), candidates.end(),

                    lambda);

                return { parent1\_idx, parent2\_idx };

            }

        };

    }

*namespace* crossover{

*namespace* RGA{

*class* BLX{

*public:*

*double* alpha;

                BLX(*double* *alpha* = 0.5):alpha(*alpha*){};

*void* doit(const std::vector<*double*>& *parent1*, const std::vector<*double*>& *parent2*, std::vector<*double*>& *child1*, std::vector<*double*>& *child2*){

                    //std::vector<double> c1(a.size()), c2(a.size());

*int* n = *parent1*.size();

*child1*.resize(n);

*child2*.resize(n);

                    for(size\_t i = 0; i < *parent1*.size(); i++){

*double* cmin = std::min(*parent1*[i],*parent2*[i]);

*double* cmax = std::max(*parent1*[i],*parent2*[i]);

*double* di = cmax - cmin;

*child1*[i] = cmin + random\_double(-alpha\*di, (1+alpha)\*di);

*child2*[i] = cmin + random\_double(-alpha\*di, (1+alpha)\*di);

                    }

                    //return {c1, c2};

                }

            };

*class* SBX{

*public:*

*double* eta;

                SBX(*double* *eta*=2):eta(*eta*){};

*void* doit(const std::vector<*double*>& *parent1*,const std::vector<*double*>& *parent2*,

                                std::vector<*double*>& *child1*, std::vector<*double*>& *child2*){

*int* n = *parent1*.size();

*child1*.resize(n);

*child2*.resize(n);

                    for(*int* i = 0; i< n; i++){

*double* u = random\_double(0.0,1.0);

*double* beta;

                        if (u <= 0.5) {

                            beta = pow(2 \* u/\* + (1 - 2 \* u) \* pow(1 - u, eta)\*/, 1.0 / (eta + 1.0));

                        } else {

                            beta = pow(1 / (2 \* (1 - u) /\*+ 2 \* (u - 0.5) \* pow(1 - u, eta)\*/), 1.0 / (eta + 1.0));

                        }

*child1*[i] = 0.5 \* ((1 + beta) \* *parent1*[i] + (1 - beta) \* *parent2*[i]);

*child2*[i] = 0.5 \* ((1 - beta) \* *parent1*[i] + (1 + beta) \* *parent2*[i]);

                    }

                }

            };

        }

    }

*namespace* mutation{

*class* non\_uniform{

*public:*

*double* mutation\_rate;

*double* min\_value;

*double* max\_value;

*double* eta;

            non\_uniform(*double* *min\_value* = 0.0, *double* *max\_value* = 1.0, *double* *eta* = 5.0)

                :  min\_value(*min\_value*), max\_value(*max\_value*), eta(*eta*) {}

*void* doit(std::vector<*double*>& *genotype*, size\_t *generation*, size\_t *max\_generations*) {

*int* idx = random\_long(0, *genotype*.size() - 1);

*int* dir = random\_long(0, 1);

*double* diff;

*double* r = random\_double(0.0, 1.0);

*double* delta = (1-pow(r, pow(1-*generation*/*max\_generations*, eta)));

                if(dir){

                    diff = -(*genotype*[idx] - min\_value);

                }else{

                    diff = max\_value - *genotype*[idx];

                }

*genotype*[idx] += delta \* diff;

            }

        };

    }

*namespace* genome{

*class* BGA{

            std::vector<*bool*> gene;

        };

*template*<size\_t size>

*class* RGA{

*public:*

            std::vector<*double*> genotype;

            RGA():genotype(size){};

            RGA(const std::vector<*double*>& *genes* ):genotype(*genes*){};

            //RGA(size\_t size):genotype(size){};

            std::vector<*double*>& get(){

                return genotype;

            }

*void* rnd(const std::vector<*double*>& *min*, const std::vector<*double*>& *max*){

                for(*int* i = 0; i< size; i++){

                    genotype[i] = random\_double(*min*[i],*max*[i]);

                }

            }

        };

    }

*template* <*typename* T>

*class* population{

*public:*

        population(): popul(){};

        population(size\_t *size*): popul(*size*){};

        std::vector<std::pair<T,*double*>> get(){

            return popul;

        }

        size\_t size(){

            return popul.size();

        }

        T& at(size\_t *idx*){

            return popul[*idx*].first;

        }

*double*& fit\_at(size\_t *idx*){

            return popul[*idx*].second;

        }

*void* push\_back(T *elem*){

            popul.push\_back({*elem*,0.0});

        }

*void* append\_range(population& *other*){

*auto* other\_content  =*other*.get();

*int* old\_size = this->size();

            this->resize(old\_size+*other*.size());

            for(*int* i = old\_size; i< this->size();i++){

                popul[i] = other\_content[i-old\_size];

            }

            //popul.append\_range(other.get());

        }

*void* fill(std::vector<*double*> *min*,std::vector<*double*> *max*){

            for(*int* i = 0;i<popul.size();i++){

                popul[i].first.rnd(*min*,*max*);

                popul[i].second = 0;

            }

        }

*void* resize(size\_t *size*){

            popul.resize(*size*);

        }

*void* clear(){

            popul.clear();

        }

        std::pair<T,*double*> best(gen::reproduction::EXTREMUM *dir* = gen::reproduction::EXTREMUM::MAX){

            size\_t best\_idx  = -1;

*double* best\_fitness=*dir* == gen::reproduction::EXTREMUM::MAX ? std::numeric\_limits<*double*>::min() : std::numeric\_limits<*double*>::max();

            for(*int* i = 0;i<this->size();i++){

*bool* is\_better = *dir* == gen::reproduction::EXTREMUM::MAX ? popul[i].second > best\_fitness : popul[i].second < best\_fitness;

                if(is\_better){

                    best\_idx = i;

                    best\_fitness = popul[i].second;

                }

            }

            return popul[best\_idx];

        }

*void* print(){

            for(*int* i = 0;i<this->size();i++){

               std::cout<<" ("<<popul[i].first.get()[0]<<","<<popul[i].first.get()[1]<<"):"<<popul[i].second<<std::endl;

            }

        }

*void* sort(gen::reproduction::EXTREMUM *dir* = gen::reproduction::EXTREMUM::MAX){

*auto* comp = [*dir*](std::pair<T,*double*> *a*,std::pair<T,*double*> *b*){return *dir*== gen::reproduction::EXTREMUM::MAX ? *a*.second>*b*.second:*a*.second<*b*.second;};

            std::sort(popul.begin(),popul.end(), comp);

        }

        std::vector<*double*> get\_fit(){

            std::vector<*double*> res;

            for(*auto*& elem: popul){

                res.push\_back(elem.second);

            }

            return res;

        }

*void* clamp(std::vector<*double*> *min*, std::vector<*double*> *max*){

            for(*auto*& elem: popul){

                for(*int* i = 0;i<elem.first.get().size();i++){

                    elem.first.get()[i] = std::clamp(elem.first.get()[i],*min*[i],*max*[i]);

                }

            }

        }

*public:*

        std::vector<std::pair<T,*double*>> popul;

    };

*template*<*typename* GeneType,*typename* ReproductionPolicy, *typename* MutationPolicy, *typename* CrossoverPolicy>

*class* GA{

*public:*

        gen::population<GeneType> population;

        gen::population<GeneType> new\_population;

        ReproductionPolicy reproduction\_p;

        MutationPolicy mutation\_p;

        CrossoverPolicy crossover\_p;

        size\_t population\_size;

        size\_t current\_step=0;

        size\_t max\_step=100;

        std::vector<*double*> min\_borders;

        std::vector<*double*> max\_borders;

        std::function<*double*(const std::vector<*double*>)> algo;

        gen::reproduction::EXTREMUM dir;

        GA<GeneType,ReproductionPolicy,MutationPolicy,CrossoverPolicy>(GeneType *gt*, ReproductionPolicy *rp*, MutationPolicy *mp*, CrossoverPolicy *cp*, gen::reproduction::EXTREMUM *dir* = gen::reproduction::EXTREMUM::MAX){

            population\_size = 20;

            population = gen::population<GeneType>(population\_size);

            new\_population = gen::population<GeneType>(population\_size);

            this->dir = *dir*;

            reproduction\_p =*rp*;

            reproduction\_p.dir = *dir*;

            mutation\_p = *mp*;

            crossover\_p= *cp*;

        }

*void* set\_population\_size(*int* *size*){

            this->population\_size=*size*;

            population.resize(population\_size);

            new\_population.resize(population\_size);

        }

*void* set\_algo(std::function<*double*(std::vector<*double*>)> *algo*){

                this->algo = *algo*;

        }

*void* fill(std::vector<*double*> *min*, std::vector<*double*> *max*){

            population.fill(*min*,*max*);

            this->min\_borders = *min*;

            this->max\_borders = *max*;

        }

*void* calculate\_fitness(){

            if (!algo) {

                throw std::runtime\_error("Fitness algorithm (algo) not set.");

            }

            for(*int* i =0; i< population.size(); i++){

                population.fit\_at(i) = algo(population.at(i).get());

            }

        }

        std::pair<GeneType&,GeneType&> reproduct(){

            std::pair<size\_t, size\_t> parents\_idx = reproduction\_p.doit(population.get\_fit());

            return { population.at(parents\_idx.first), population.at(parents\_idx.second) };

        }

*void* crossover(){

*int* n = population.size();

             std::vector<std::vector<*double*>> v;

            for(*int* i = 0; i< n/2; i++){

                std::vector<*double*> child1, child2;

                const std::pair<GeneType&,GeneType&> parents = reproduct();

                v.push\_back(parents.first.get());

                 v.push\_back(parents.first.get());

                crossover\_p.doit(parents.first.get(),parents.second.get(),child1,child2);

                new\_population.at(i\*2) = GeneType(child1);

                new\_population.at(i\*2+1)=GeneType(child2);

            }

        }

*void* mutate(){

            for(*int* i = 0; i< population.size();i++){

                mutation\_p.doit(new\_population.at(i).get(),current\_step,max\_step);

            }

        }

*void* reduce(){

            population.clamp(min\_borders,max\_borders);

            calculate\_fitness();

            population.sort(dir);

            //population.print();

            population.resize(population\_size);

        }

*void* doit(){

            calculate\_fitness();

            crossover();

            mutate();

            population.append\_range(new\_population);

            reduce();

            current\_step++;

        }

*void* doit(size\_t *n*){

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

                doit();

            }

        }

        std::vector<std::pair<GeneType,*double*>> get\_population(){

            std::vector<std::pair<GeneType,*double*>> ret;

            for(*int* i = 0; i< population.size();i++){

                ret.push\_back({population.at(i),population.fit\_at(i)});

            }

            return ret;

        }

    };}

}

#include <iostream>

#include <vector>

#include <functional>

#include <fstream>

#include <string>

#include <execution>

#include <algorithm>

#include "../libs/gplot++.h"

#include "genetic.hpp"

*template* <*typename* T>

*void* save(const std::string &*filename*,  std::vector<std::pair<T,*double*>> *args*) {

    std::ofstream os(*filename*);

    for(*auto*& arg: *args*){

        for(*auto*& x: arg.first.get()){

            os<<x<<" ";

        }

        os<<arg.second<<std::endl;

    }

}

*void* draw(const std::string &*filename*){

    Gnuplot plt{};

    std::string result = R"(

set grid

set xrange [-500:500]

set yrange [-500:500]

set zrange [-1000:1000]

set isosamples 100

set samples 100

set contour base

# Define the function

f(x, y) = (-x) \* sin(sqrt(abs(x))) + (-y) \* sin(sqrt(abs(y)))

set cntrparam levels incremental -1000, 100, 1000  # Define contour steps from -1000 to 1000 with step 100

# Plot the function in 3D and add the contour plot

splot f(x, y) with lines , ")"+*filename*+R"(" with points  pt 7 ps 1.8)" ;    plt.sendcommand(result);

}

*template*<*typename* G, *typename* R,*typename* M,*typename* C>

*void* ui(gen::GA<G,R,M,C> &*ga*){

*int* population\_size;

    std::cout<<"Choose population size"<<std::endl;

    std::cin>>population\_size;

*ga*.set\_population\_size(population\_size);

*ga*.fill({-500,-500},{500,500});

*ga*.calculate\_fitness();

    std::cout<<"Initial state"<<std::endl;

*auto* best = *ga*.population.best(gen::reproduction::EXTREMUM::MIN);

    std::cout<<"Best: ("<<best.first.get()[0]<<","<<best.first.get()[1]<<"):"<<best.second<<std::endl;

*auto* result = *ga*.get\_population();

    save<gen::genome::RGA<2>>("result.txt",result);

    draw("result.txt");

*int* n = 0;

    while(true){

        std::cout<<"Enter number of iterations or 0 to exit"<<std::endl;

        std::cin>>n;

        if(n==0) break;

*ga*.doit(n);

        best = *ga*.population.best(gen::reproduction::EXTREMUM::MIN);

        std::cout<<"Best: ("<<best.first.get()[0]<<","<<best.first.get()[1]<<"):"<<best.second<<std::endl;

        result = *ga*.get\_population();

        save<gen::genome::RGA<2>>("result.txt",result);

        draw("result.txt");

    }

    std::cout<<"Exiting..."<<std::endl;

}

// Сохранение результата одного эксперимента

*void* save\_results(const std::string &*filename*, *int* *n*, *double* *best\_avg\_result*, *double* *avg\_of\_avg\_result*) {

    std::ofstream os(*filename*, std::ios\_base::app); // Открываем в режиме добавления

    os << *n* << " " << *best\_avg\_result* << " " << *avg\_of\_avg\_result* << std::endl;

}

// Построение графика через gnuplot

*void* draw\_results(const std::string &*filename*) {

    Gnuplot plt{};

    std::string cmd = R"(

set grid

set xlabel 'n'

set ylabel 'Fitness'

set title 'Best and Average Fitness vs n'

plot ')" + *filename* + R"(' using 1:2 with linespoints title 'Best Average Fitness', \

     ')" + *filename* + R"(' using 1:3 with linespoints title 'Average of Average Fitness'

)";

    plt.sendcommand(cmd);

}

// Функция для одного эксперимента

*template*<*typename* G, *typename* R, *typename* M, *typename* C>

std::pair<*double*, *double*> run\_experiment(gen::GA<G, R, M, C> &*ga*, *int* *epch*) {

    // Запуск ГА на epch шагов

*ga*.doit(*epch*);

    // Получение лучшего результата

*auto* best\_pair = *ga*.population.best(gen::reproduction::EXTREMUM::MIN);

*double* best\_fitness = best\_pair.second;

    // Вычисление среднего значения по всей популяции

*double* avg\_fitness = 0.0;

    for (const *auto* &individual : *ga*.get\_population()) {

        avg\_fitness += individual.second;

    }

    avg\_fitness /= *ga*.get\_population().size();  // Среднее значение популяции

    return {best\_fitness, avg\_fitness};

}

// Основная функция для многократного запуска экспериментов с разными n

*void* run\_experiments(*int* *epch*, const std::vector<*int*> &*n\_values*, const std::string &*output\_filename*, const *int* *population\_size*) {

*auto* schwefel = [](const std::vector<*double*> &*x*) -> *double* {

*double* sum = 0.0;

        const *double* constant = 418.9829;

        for (*double* xi : *x*) {

            sum -= xi \* sin(sqrt(fabs(xi)));

        }

        return sum;

    };

    std::ofstream os(*output\_filename*);

    os.close();

    // Перебираем разные значения n

    for (*int* n : *n\_values*) {

*double* total\_best = 0.0;  // Для усреднения лучших значений

*double* total\_avg = 0.0;   // Для усреднения средних значений

        const *int* iterations = 500;  // Количество запусков эксперимента для усреднения

    const *double* extremum = 2 \* 418.9829;

        // Многократные запуски для усреднения

        for (*int* i = 0; i < iterations; ++i) {

            gen::GA ga(gen::genome::RGA<2>(),

                   gen::reproduction::roulette{},

                   gen::mutation::non\_uniform{-500, 500},

                   gen::crossover::RGA::SBX{n},

                   gen::reproduction::EXTREMUM::MIN);

            ga.set\_algo(schwefel);

            ga.set\_population\_size(*population\_size*);

            ga.fill({-500, -500}, {500, 500});

            ga.calculate\_fitness();

            // Получаем пару значений: лучший результат и среднее значение популяции

*auto* [best, avg] = run\_experiment(ga, *epch*);

            total\_best += ((best+extremum)\*(best+extremum)/iterations);  // Суммируем лучшие результаты

            total\_avg += ((avg+extremum)\*(avg+extremum)/iterations);    // Суммируем средние значения популяции

        }

        // Средний лучший результат и средний результат по популяции

        //double best\_avg\_result = total\_best / iterations;

       // double avg\_of\_avg\_result = total\_avg / iterations;

        //best\_avg\_result += extremum;

        //avg\_of\_avg\_result += extremum;

        // Сохранение результатов

        save\_results(*output\_filename*, n, sqrt(total\_best), sqrt(total\_avg));

    }

}

*int* main(){

*auto* schwefel = [](const std::vector<*double*>& *x*) -> *double* {

*double* sum = 0.0;

        const *double* constant = 418.9829;

        for (*double* xi : *x*) {

            sum -= xi \* sin(sqrt(fabs(xi)));

        }

        return sum;

    };

    std::cout<<"Menu:"<<std::endl;

    std::cout<<"\t1)Steps"<<std::endl;

    std::cout<<"\t2)Experiment"<<std::endl;

    std::cout<<"\t3)Exit"<<std::endl;

*int* menu = 0;

*int* n =0;

    std::cin>>menu;

    switch(menu){

        case 1:{

             n = 2;

            std::cout<<"Choose N value for SBX "<<std::endl;

            std::cin>>n;

            gen::GA ga(gen::genome::RGA<2>(),gen::reproduction::roulette{},gen::mutation::non\_uniform{-500,500},gen::crossover::RGA::SBX{n},gen::reproduction::EXTREMUM::MIN);

            ga.set\_algo(schwefel);

            ui(ga);

        }

            break;

        case 2:{

*int* epch = 0;  // Количество поколений для каждого эксперимента

            std::cout<<"Choose number of epochs"<<std::endl;

            std::cin>>epch;

            std::cout<<"choose population size"<<std::endl;

            std::cin>>n;

            std::vector<*int*> n\_values = {2, 3, 4, 5};  // Значения n

            std::string output\_filename = "fitness\_results.txt";  // Файл для сохранения результатов

            run\_experiments(epch, n\_values, "fitness\_results.txt", n);

            draw\_results(output\_filename);

        }

            break;

        case 3:

            return 0;

    }

    return 0;

}

**4. Исследование**

С помощью функции run\_experiments, я исследовал зависисмость ошибки от размера популяции, числа эпох и параметра n.

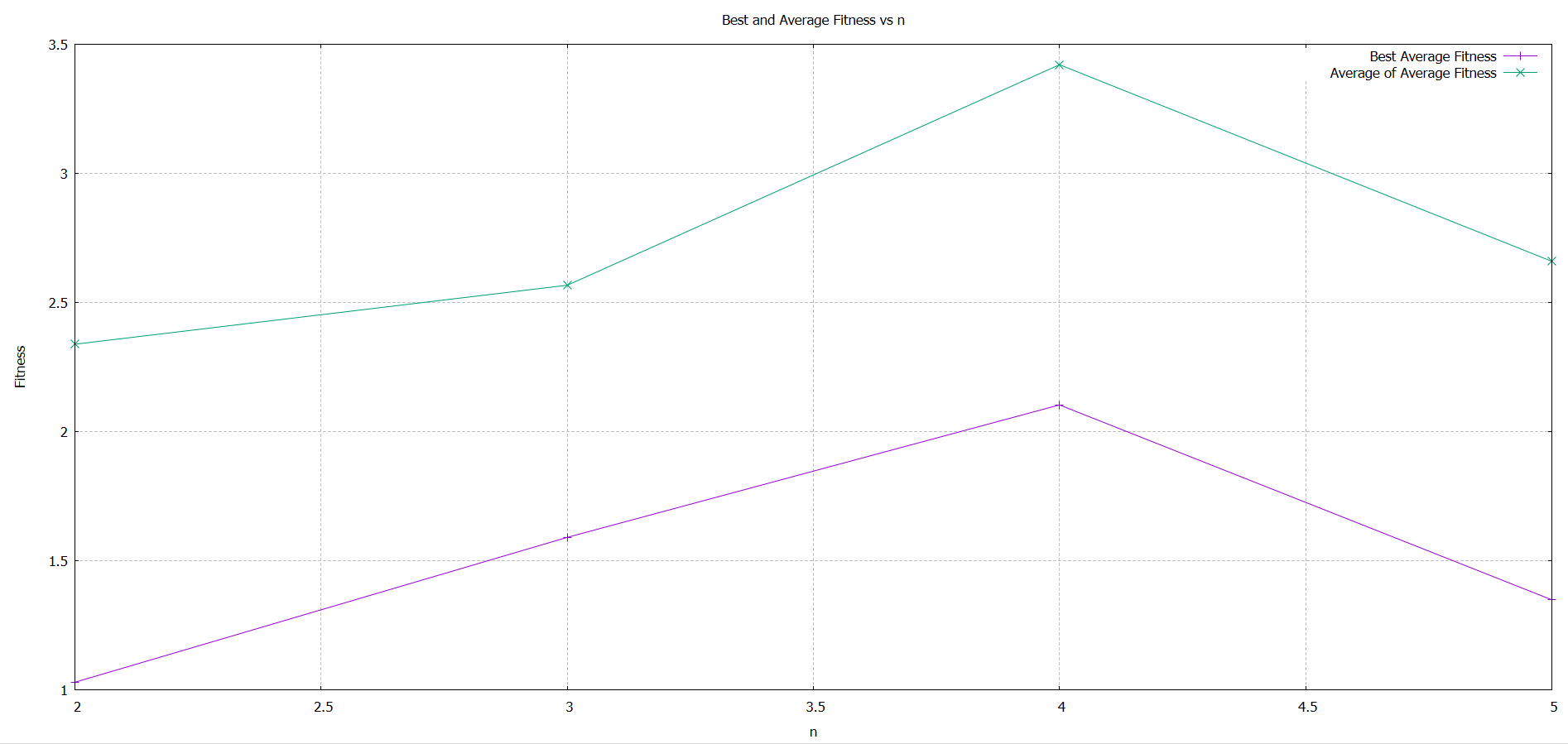
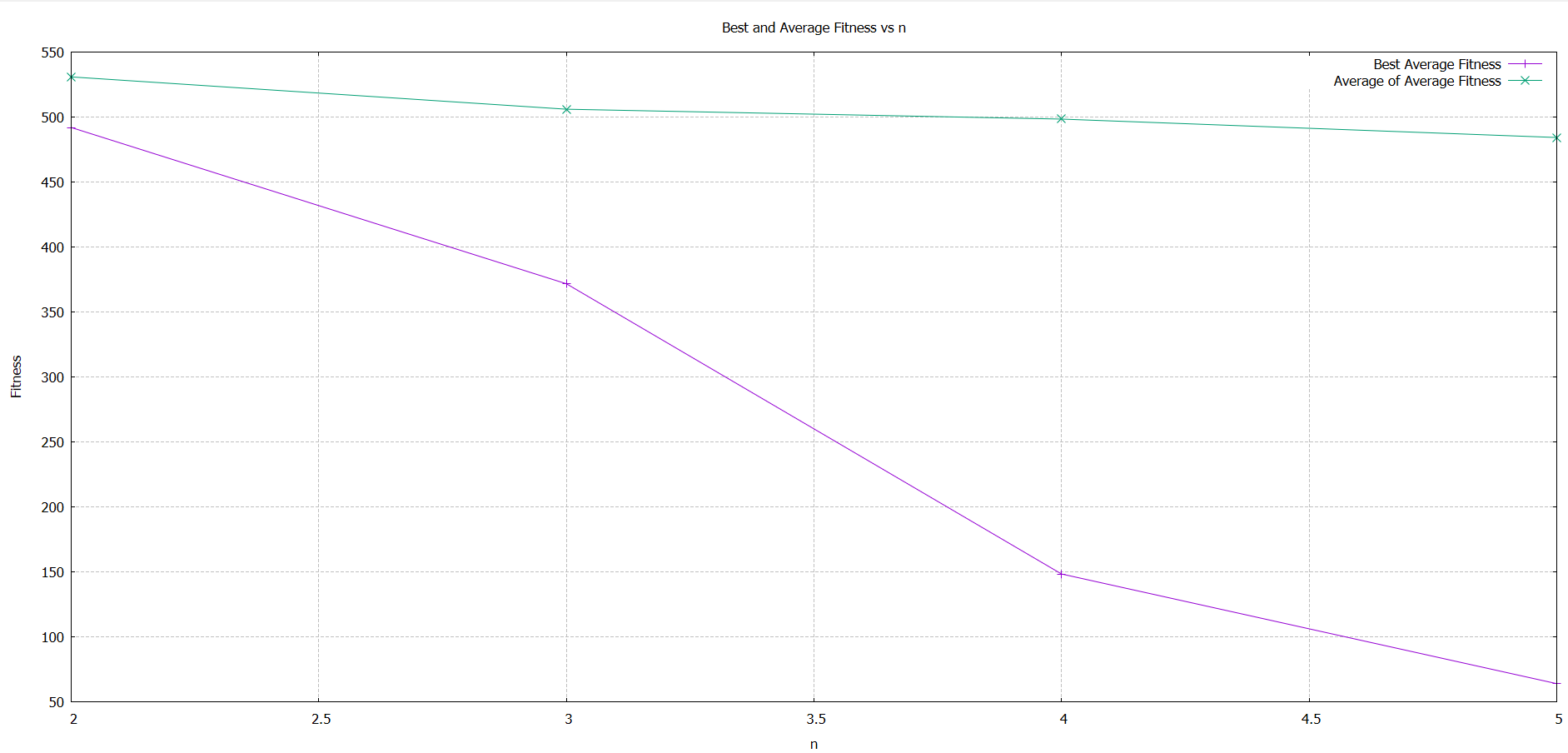


График ошибки для популяции мощностью 10 и 100 эпохами

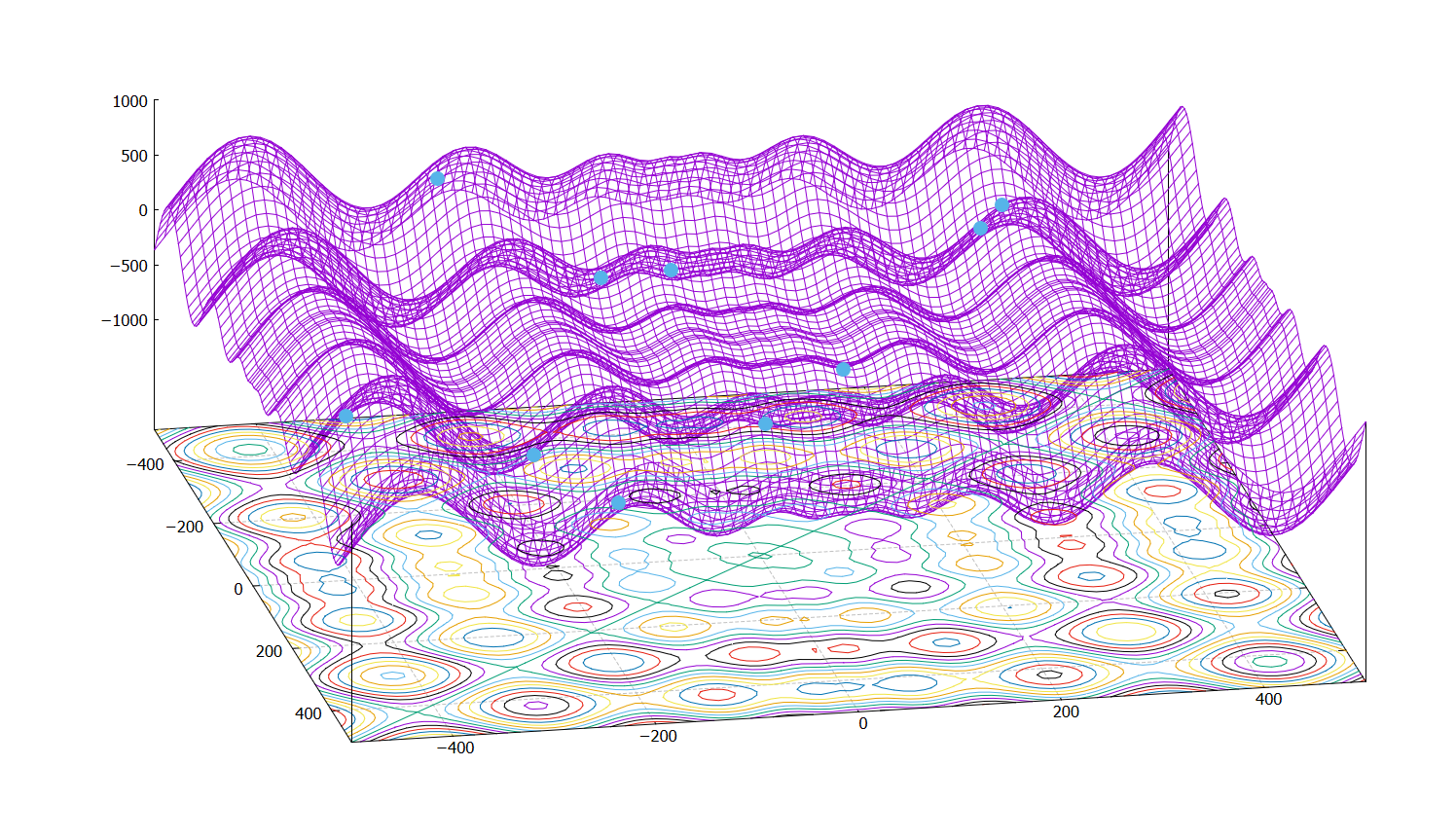
Для малой популяции заметно, что чем меньше n, тем точнее результат

  
График ошибки для популяции мощностью 100 и 10 эпохами

Для большой же популяции наоборот, так как для в ней больший шанс что точки изначально попадут недалеко от экстремума

**5. Выполнение симуляции**

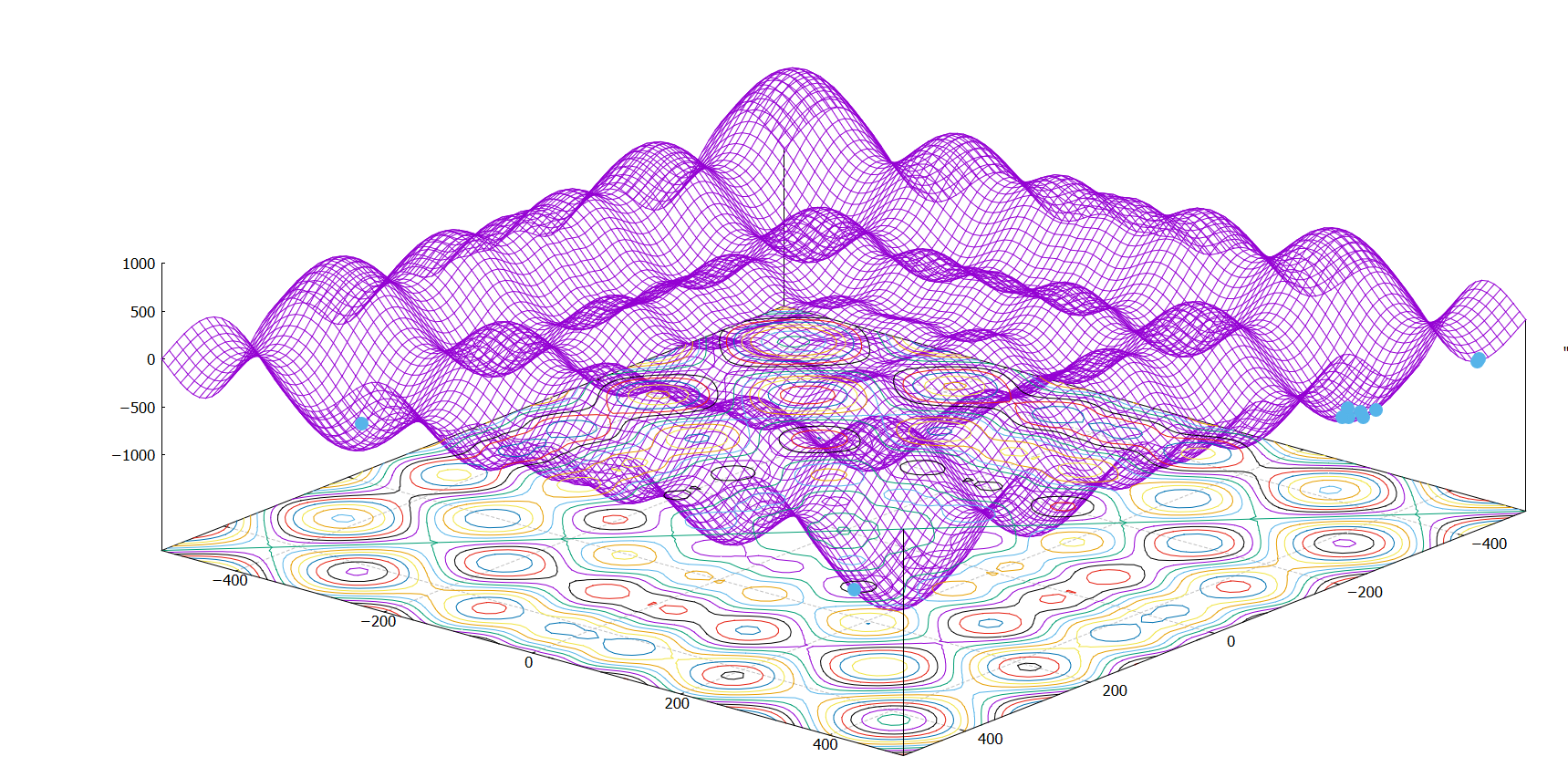
**Шаг 1**

****

Начальная популяция

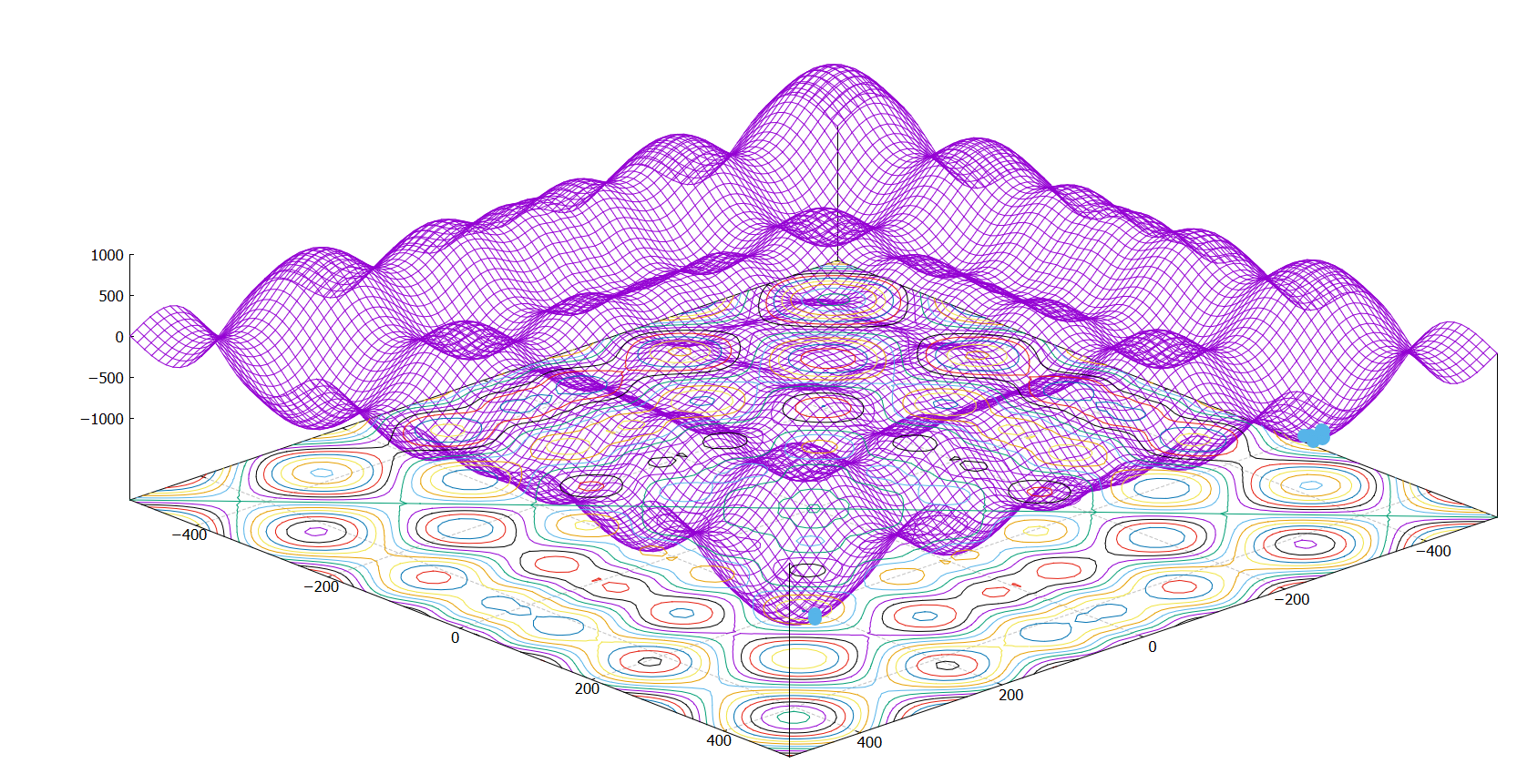
Лучшая точка (213.098,-264.401):-329.629

**Шаг 5**

****Популяция спустя 5 шагов, выделились два экстремума

Лучшая точка (-311.024,419.574):-710.139

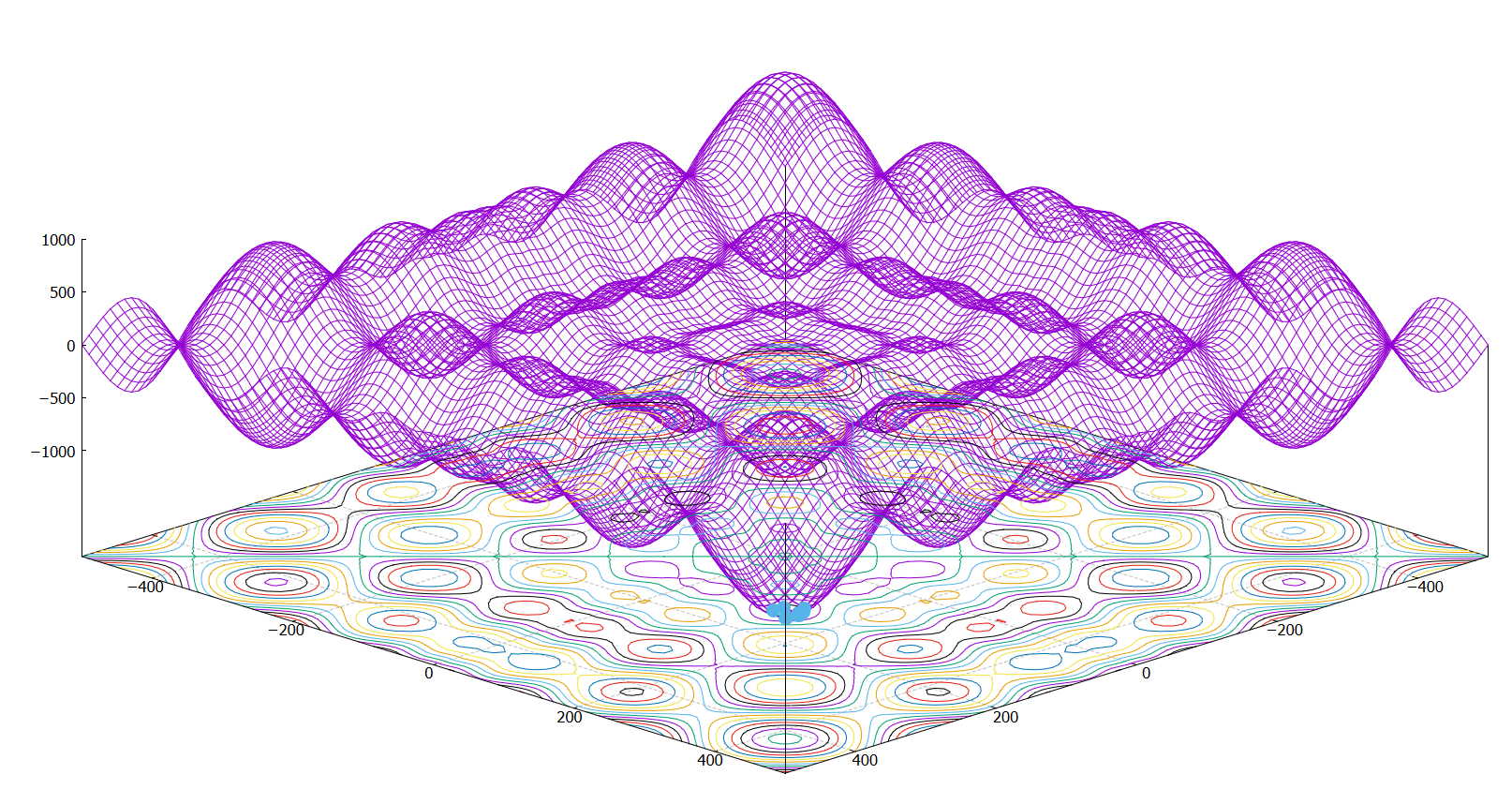
**Шаг 10**

****Популяция спустя 10 шагов

Так же два экстремума, но в глобальном появляется больше точек.

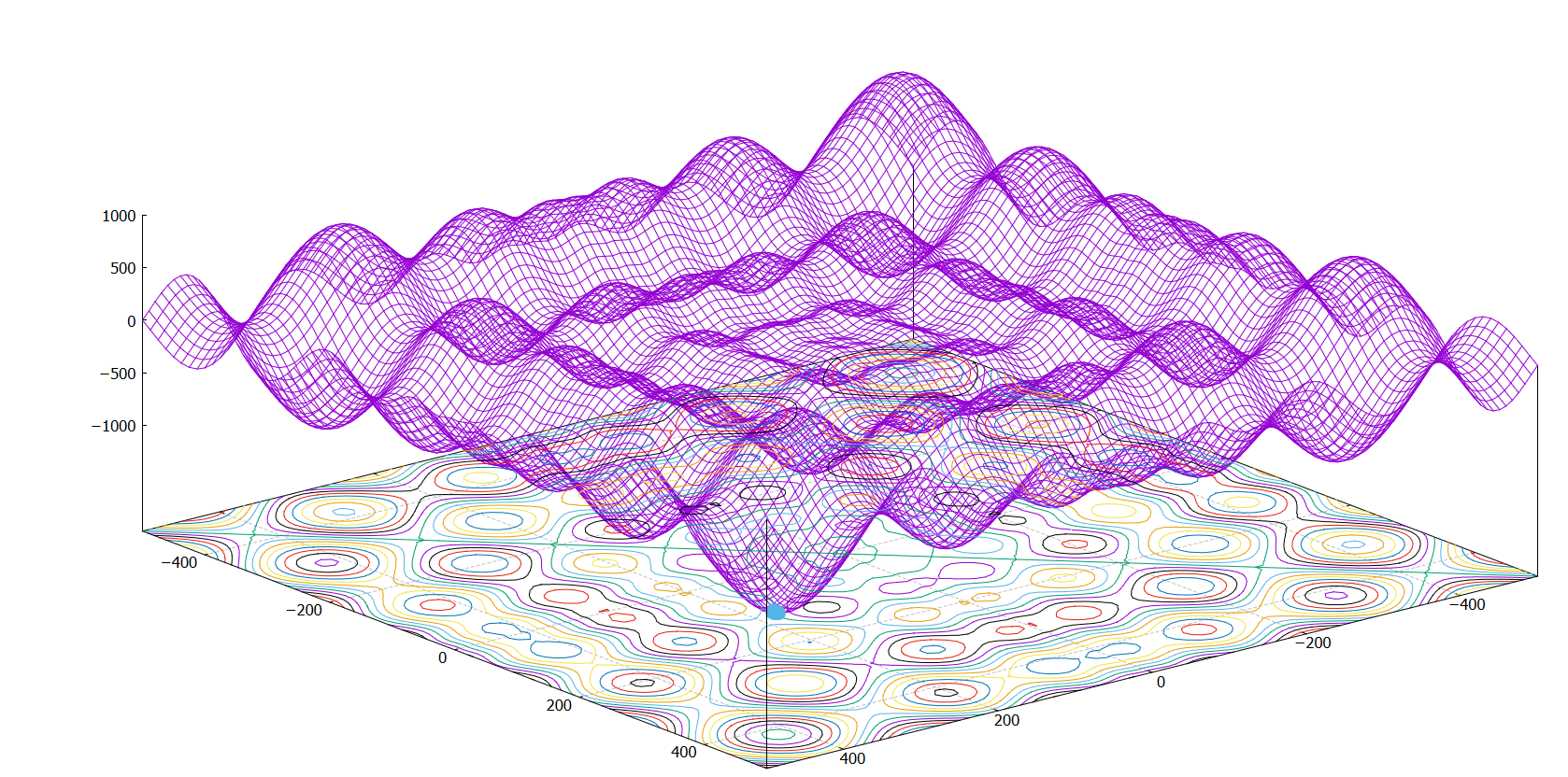
Лучшая точка (400.226,430.678):-773.401

**Шаг 15**

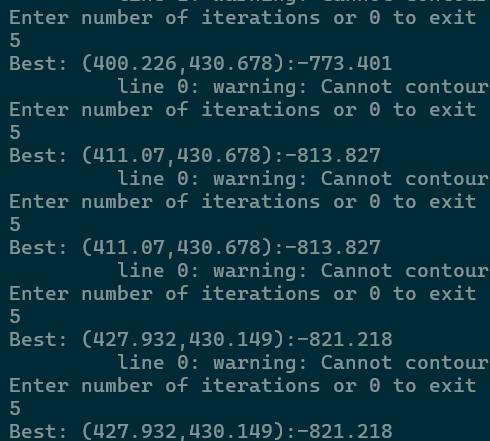
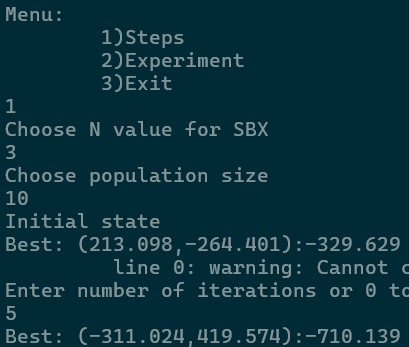
****Популяция спустя 15 шагов

Лучшая точка— (427.932,430.149):-821.218

**Шаг 50**

****Популяция спустя 50 шагов

Лучшая точка— (422.8,422.281):-837.325

  
Скриншот вывода программы во время поискаЗаканчиваем поиск, так как большая часть точек находится в одном месте.

Точка экстремума — (420.9687, 420.9687)

Ошибка —(1.9, 1.3): 0.6

**6. Контрольный вопрос**

**Дайте понятие экстремума и оптимума целевой функции.**

Экстремум – максимальное или минимальное значение функции. Оптимум – аргументы функции при которых этот экстремум достигается.