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 |
| «*Простой генетический алгоритм*» |
| 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. Разработать простой генетический алгоритм для нахождения оптимума заданной по варианту функции одной переменной (таб. 1.1). Вид экстремума:

2. Исследовать зависимость времени поиска, числа поколений (генераций), точности нахождения решения от основных параметров генетического алгоритма: - число особей в популяции - вероятность кроссинговера, мутации.

3. Вывести на экран график данной функции с указанием найденного экстремума для каждого поколения.

4. Сравнить найденное решение с действительным.

Вариант 7





**2. Основные сведения из теории**

ГА используют принципы и терминологию, заимствованные у биологической науки – генетики. В ГА каждая особь представляет потенциальное решение некоторой проблемы.

ГА берет множество параметров оптимизационной проблемы и кодирует их последовательностями конечной длины в некотором конечном алфавите (в простейшем случае двоичный алфавит «0» и «1») . Предварительно простой ГА случайным образом генерирует начальную популяцию стрингов (хромосом). Затем алгоритм генерирует следующее поколение (популяцию), с помощью трех основных генетических операторов: 1) Оператор репродукции (ОР); 2) Оператор скрещивания ( кроссинговера, ОК); 3) Оператор мутации (ОМ). Генетические операторы являются математической формализацией приведенных выше трех основополагающих принципов Дарвина, Менделя и де Вре естественной эволюции. ГА работает до тех пор, пока не будет выполнено заданное количество поколений (итераций) процесса эволюции или на некоторой генерации будет получено заданное качество или вследствие преждевременной сходимости при попадании в некоторый локальный оптимум.

**3. Листинг программы**

#include <bitset>

#include <random>

#include <cmath>

#include <iostream>

#include <functional>

#include <iterator>

#include <vector>

#include <algorithm>

#include <thread>

#include <mutex>

#include <chrono>

#include <fstream>

#include <array>

#include <atomic>

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

using *namespace* std;

atomic<*int*> count\_joined(0);

random\_device rnd;

mt19937 gen(rnd());

*enum* *class* OptimizationType {

    MINIMIZATION,

    MAXIMIZATION

};

*double* calculate\_fitness(*double* *value*, OptimizationType *optimization\_type*, std::function<*double*(*double*)> *function*) {

*double* fitness = *function*(*value*);

    if (*optimization\_type* == OptimizationType::MINIMIZATION) {

        return 1.0 / (fitness + 1e-10);  // Adding a small value to avoid division by zero

    } else {

        return fitness;

    }

}

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

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

    return distrib(gen);

}

*double* randomd(*double* *min*, *double* *max*){

    std::uniform\_real\_distribution<*double*> distrib(*min*, *max*);

    return distrib(gen);

}

*template*<*int* min, *int* max,*int* dad>

*class* Population{

*public:*

    constexpr Population(){}

*double* get\_best\_point(){

*auto* sort\_func = [this](Genome *a*, Genome *b*) {return  function(pos\_to\_real(*a*.getGenome())) > function(pos\_to\_real(*b*.getGenome())); };

        vector<Genome> sorted\_pop = population;

        sort(sorted\_pop.begin(),sorted\_pop.end(), sort\_func);

        return pos\_to\_real(sorted\_pop.front().getGenome());

    }

*double* get\_genome(*int* *i*){

        return pos\_to\_real(population[*i*].getGenome());

    }

*void* setFunction(function<*double*(*double*)> *fun*){

        function = *fun*;

    }

*void* fill( *int* *count*){

        assert(count%2 == 0);

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

            Genome g;

            g.setGenome(generate());

            population.push\_back(g);

        }

    }

*void* step(){

            reproduction();

            crossingover();

            mutation();

            //printp();

    }

*void* doSteps(*int* *n*){

        for(*int* i=0;i<*n*;i++){

            step();

        }

    }

*void* draw(){

        Gnuplot plt{};

        std::vector<*double*> x,y;

        for(*double* i = min; i<max;i+=0.001){

            x.push\_back(i);

            y.push\_back(function(i));

        }

        std::vector<*double*> px,py;

        for(*auto* pop: population){

            px.push\_back(pos\_to\_real(pop.gene.to\_ulong()));

            py.push\_back(function(px.back()));

        }

        plt.plot(x, y, "Dataset #1", Gnuplot::LineStyle::LINES);

        plt.plot(px, py, "Dataset #2", Gnuplot::LineStyle::POINTS);

        plt.show();

    }

*void* setPCross(*double* *p*){probability\_crossingover = *p*;}

*void* setPMut(*double* *p*){probability\_mutation = *p*;}

*private:*

*class* Genome;

    static constexpr *long* lenght(){return static\_cast<*long*>(log2((max-min)\*pow(10,dad)))+1;};

    static constexpr *long* n(){return static\_cast<*long*>(powl(2,lenght()));}

    OptimizationType optimization\_type = OptimizationType::MAXIMIZATION;

    std::function<*double*(*double*)> function;

    vector<Genome>  population = {};

    vector<Genome> mid\_population  ={};

    //Gnuplot plt{};

*double* probability\_crossingover = 0.5d;

*double* probability\_mutation = 0.01d;

*private:*

*double* pos\_to\_real(*long* *pos*){

        return *pos*\*(max-min)/static\_cast<*double*>(n()-1)+min;

    }

*long* generate(){

        return random(0,n()-1);

    }

*void* reproduction(){

        vector<*double*> fitness  ={};

        vector<*double*> probabilities  ={};

*double* fitness\_min = numeric\_limits<*double*>::max();

        for(*auto* gene: population){

*double* fit = calculate\_fitness(pos\_to\_real(gene.getGenome()), optimization\_type, function);

            fitness.push\_back(fit);

            if (fit < fitness\_min) {

                fitness\_min = fit;

            }

        }

*double* normalized\_min = fitness\_min < 0 ? -fitness\_min : 0;

        for (*auto*& fit : fitness) {

            fit += normalized\_min;

        }

*double* total\_fitness = accumulate(fitness.begin(), fitness.end(), 0.0);

        for(*auto* fit: fitness){

            probabilities.push\_back(fit/total\_fitness);

        }

        mid\_population.clear();

        vector<*double*> cumulative\_probabilities(probabilities.size());

        partial\_sum(probabilities.begin(), probabilities.end(), cumulative\_probabilities.begin());

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

*double* random\_value = randomd(0.0,1.0);

*auto* it = lower\_bound(cumulative\_probabilities.begin(), cumulative\_probabilities.end(), random\_value);

            size\_t index = distance(cumulative\_probabilities.begin(), it);

            mid\_population.push\_back(population[index]);

        }

    }

*void* crossingover(){

        vector<Genome> a\_half  ={};

        vector<Genome> b\_half  ={};

*long* a,b;

        while(mid\_population.size()>0){

            do{

                a = random(0,mid\_population.size()-1);

                b = random(0,mid\_population.size()-1);

            }while(a==b);

*auto* a\_it = std::next(mid\_population.begin(), a);

*auto* b\_it = std::next(mid\_population.begin(), b);

            a\_half.push\_back(\*a\_it);

            b\_half.push\_back(\*b\_it);

            mid\_population.erase(a\_it);

            mid\_population.erase(b\_it);

        }

        //a\_half.push\_back(mid\_population[0]);

        //b\_half.push\_back(mid\_population[1]);

        for(*int* i = 0; i < a\_half.size();i++){

*long* idx =  random(1,lenght()-1);

*double* rand\_v  = randomd(0.0d, 1.0d);

            if(rand\_v < probability\_crossingover){

                a\_half[i].swap(b\_half[i],idx);

            }

        }

        mid\_population.insert(mid\_population.end(),a\_half.begin(),a\_half.end());

        mid\_population.insert(mid\_population.end(),b\_half.begin(),b\_half.end());

    }

*void* mutation(){

        for(*auto*& pop: mid\_population){

*double* rand\_v = randomd(0.0d, 1.0d);

            if(rand\_v < probability\_mutation){

*long* idx =  random(0,lenght()-1);

                pop.gene.flip(idx);

            }

        }

        population.clear();

        for(*auto* pop: mid\_population){

            population.push\_back(pop);

        }

        //mid\_population.clear();

    }

*private:*

*class*  Genome{

*public:*

        bitset<lenght()> gene;

*void* setGenome(*unsigned* *long* *value*){

            gene = bitset<lenght()>(*value*);

        }

*long* getGenome(){

            return gene.to\_ulong();

        }

*void* flip(*int* *idx*){

            gene.flip(*idx*);

        }

*void* swap(Genome& *other*, *int* *end\_idx*){

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

*bool* tmp = gene[i];

            this->gene[i] = *other*.gene[i];

*other*.gene[i] = tmp;

            }

        }

    };

*public:*

*void* printp(){

        for(*auto* p: population){

            cout<<p.getGenome()<<" ";

        }

        cout<<endl;;

    }

*void* printm(){

        for(*auto* p: mid\_population){

            cout<<p.gene.to\_ulong()<<" ";

        }

        cout<<endl;

    }

};

vector<*int*> generate\_population\_range(*int* *min*, *int* *max*) {

    vector<*int*> populations;

    for (*int* i = *min*; i <= *max*; i += 2) {

        populations.push\_back(i);

    }

    return populations;

}

vector<*double*> generate\_range(*double* *low*, *double* *high*, *double* *step*) {

    vector<*double*> range;

    for (*double* value = *low*; value <= *high*; value += *step*) {

        range.push\_back(value);

    }

    return range;

}

// Worker function to run the benchmark

*void* benchmark\_worker(*int* *pc*, *double* *c*, *double* *m*, *int* *steps*, function<*double*(*double*)> *fun*, *double* *local\_extr\_v*,

                      vector<tuple<*int*, *double*, *double*, *double*,*double*>>& *local\_results*, mutex& *results\_mutex*) {

    Population<1, 10, 5> pop = {};

    pop.fill(*pc*);

    pop.setFunction(*fun*);

    pop.setPCross(*c*);

    pop.setPMut(*m*);

*int* n = 0;

*double* last\_error = 123456789;

*int* close\_points = 0;

    const *double* close\_dist = (10-1)/*double*(20);

*double* best\_point = pop.get\_best\_point();

*auto* start\_time = chrono::high\_resolution\_clock::now();

    while(n++<*steps*){

        pop.step();

        for(*int* i = 0;i<*pc*;i++){

*double* error = pop.get\_genome(i) - best\_point;

            if(fabs(error) > close\_dist\*2) {

                continue;

            }

            if(abs(error)<close\_dist){

                close\_points++;

            }

            if(close\_points>=(*pc*\*0.9)){

            break;

            }

        }

        if(close\_points>=*double*(*pc*\*0.9)+1){

            break;

        }

    }

*auto* end\_time = chrono::high\_resolution\_clock::now();

    chrono::duration<*double*> duration = end\_time - start\_time;

*double* error = abs(*fun*(pop.get\_best\_point())-*local\_extr\_v*);

    {

        lock\_guard<mutex> guard(*results\_mutex*);

        //cout<<"thread "<<pc<<c<<m<<" ended"<<endl;

*local\_results*.push\_back({*pc*, *c*, *m*,duration.count(),error});

    }

}

*void* monitor\_progress(*int* *total\_threads*) {

*int* percent\_size = *total\_threads* / 100;

    while (count\_joined.load() < *total\_threads*\*0.8) {

        cout << "Progress: " << count\_joined.load() / percent\_size << "%" << endl;

        this\_thread::sleep\_for(chrono::milliseconds(500));  // Check every 500ms

*int* r = count\_joined.load();

*int* r2=2+r;

    }

}

*long* estimate\_iterations(*int* *population\_min*, *int* *population\_max*, *int* *population\_step*,

*double* *low\_c*, *double* *high\_c*, *double* *step\_c*,

*double* *low\_m*, *double* *high\_m*, *double* *step\_m*) {

    // Calculate the number of iterations for each parameter

*long* population\_iterations = (*population\_max* - *population\_min*) / *population\_step* + 1;

*long* crossover\_iterations = static\_cast<*long*>((*high\_c* - *low\_c*) / *step\_c*) + 1;

*long* mutation\_iterations = static\_cast<*long*>((*high\_m* - *low\_m*) / *step\_m*) + 1;

    // Total iterations is the product of all iterations

    return population\_iterations \* crossover\_iterations \* mutation\_iterations;

}

vector<tuple<*int*, *double*, *double*, *double*,*double*>> gen\_benchmark(*int* *population\_min*, *int* *population\_max*,*int* *population\_step*, *double* *low\_c*, *double* *high\_c*, *double* *step\_c*,

*double* *low\_m*, *double* *high\_m*, *double* *step\_m*, *int* *steps*, function<*double*(*double*)> *fun*, *double* *fun\_extr*) {

*double* local\_extr;

    local\_extr = *fun\_extr*;

*double* local\_extr\_v = *fun*(local\_extr);

*auto* start\_time = chrono::high\_resolution\_clock::now();

    vector<tuple<*int*, *double*, *double*, *double*, *double*>> results;

    vector<thread> threads;

    mutex results\_mutex;

    thread progress\_thread(monitor\_progress, estimate\_iterations(*population\_min*, *population\_max*, *population\_step*, *low\_c*, *high\_c*, *step\_c*, *low\_m*, *high\_m*, *step\_m*));

    for( *int* pc = *population\_min*; pc<=*population\_max*; pc+=*population\_step*){

        for(*double* c = *low\_c*;c<=*high\_c*;c += *step\_c*){

            for(*double* m = *low\_m*;m<=*high\_m*;m += *step\_m*){

               threads.emplace\_back(benchmark\_worker,pc,c,m,*steps*,*fun*,local\_extr\_v,ref(results),ref(results\_mutex));

               count\_joined++;

            }

        }

    }

    for (*auto*& thread : threads) {

        thread.join();

    }

    progress\_thread.join();

*auto* end\_time = chrono::high\_resolution\_clock::now();

    chrono::duration<*double*> duration = end\_time - start\_time;

    cout << "Benchmark completed in " << duration.count() << " seconds." << endl;

    return results;

}

*template* <*typename* T>

*void* write\_to\_file(const string& *filename*, const vector<T>& *data*) {

    ofstream ofs(*filename*);

    if (!ofs) {

        cerr << "Error opening file for writing: " << *filename* << endl;

        return;

    }

    for (const *auto*& entry : *data*) {

            std::apply(

                [&ofs](*auto*&&... *args*) {

                    ((ofs << *args* << " "), ...);

                    ofs << endl;

                },

                entry);

        }

        ofs.close();

    }

*void* plot\_gnu(const string& *filename*,const vector<*int*>& *columns*, const array<string,5>& *titles*, *bool* *is\_log*=0) {

    Gnuplot plt{};

    string labels="";

    string log="";

    string plot\_type = "";

    string columns\_str = "";

    if(*columns*.size()==2){

        labels = "set xlabel '"+*titles*[*columns*[0]]+"'; set ylabel '"+*titles*[*columns*[1]]+"'; ";

        if(*is\_log*){

            log = "set logscale y; ";

        }

        plot\_type= "plot";

        columns\_str=to\_string(*columns*[0]+1)+":"+to\_string(*columns*[1]+1)+":"+to\_string(*columns*[1]+1);

    }else if(*columns*.size()==3){

        labels = "set xlabel '"+*titles*[*columns*[0]]+"'; set ylabel '"+*titles*[*columns*[1]]+"'; set zlabel '"+*titles*[*columns*[2]]+"'; ";

        if(*is\_log*){

            log = "set logscale z; ";

        }

        plot\_type= "splot";

        columns\_str=to\_string(*columns*[0]+1)+":"+to\_string(*columns*[1]+1)+":"+to\_string(*columns*[2]+1);

    }else{

        cout<<"Wrong number of columns"<<endl;

        return;

    }

    string result = "set palette defined (0 'blue', 1.00E-17 'green', 1.00E-10 'yellow',1.00E-8 'red'); "

                                   "set cblabel 'Error Value'; "

                                   +labels+" "

                                   +log

                                    +" set style fill solid 0.5 noborder; "

                                    +plot\_type+" '" + "results.txt" + "' using "+ columns\_str +  " with points  pt 1 ps 0.8 palette title '" + "GRAPH" + "'";

    plt.sendcommand(result);

}

*int* main(){

    const array<string,5> columns\_title = {"Population", "Crossingover", "Mutation","Time" ,"Error"};

    const string filename1 = "3d\_data1.txt";

    const string filename2 = "3d\_data2.txt";

    const string filename3 = "3d\_data3.txt";

*int* n=-1;

    while(n!=9){

        cout<<"Choose option:"<<endl;

        cout<<"\t1)Recalculate benchmark and save to file"<<endl;

        cout<<"\t2)Plot files"<<endl;

        cout<<"\t3)Load individual simulation"<<endl;

        cout<<"\t9)Exit"<<endl;

        cin>>n;

        switch(n){

            case 1:

            {

                const *double* time = *double*(135)/*double*(51272\*20);

*int* steps = 100;

*int* min\_pop=2;

*int* max\_pop = 500;

*int* step\_pop = 2;

*double* min\_cop=0.1;

*double* max\_cop = 0.9;

*double* step\_cop = 0.05;

*double* min\_mp=0.0001;

*double* max\_mp = 0.4;

*double* step\_mp = 0.01;

*auto* fun = [](*double* *x*) { return log(*x*)\*cos(3\**x*-15); };

*double* local\_extr =9.19424;

*long* est\_iterations = estimate\_iterations(min\_pop, max\_pop, step\_pop, min\_cop, max\_cop, step\_cop, min\_mp, max\_mp, step\_mp);

                cout<<"Estimated iterations:"<<est\_iterations<<endl;

                cout<<"Estimated time:"<<est\_iterations\*time\*steps<<"'s"<<endl;

*auto* results = gen\_benchmark(min\_pop, max\_pop, step\_pop, min\_cop, max\_cop, step\_cop, min\_mp, max\_mp, step\_mp, steps,fun,local\_extr);

                sort(results.begin(), results.end(),[](*auto* *a*,*auto* *b*){return get<3>(*a*)<get<3>(*b*);});

*int* cnt = 0;

                cout<<"Top 20 results:"<<endl;

                for (const *auto*& result : results) {

                    cout << "Population: " << get<0>(result)

                        << ", Crossover: " << get<1>(result)

                        << ", Mutation: " << get<2>(result)

                        << ", Error: " << get<3>(result) << endl;

                    if(cnt++>=20){

                        break;

                    }

                }

                write\_to\_file<tuple<*int*, *double*, *double*, *double*,*double*>>("results.txt", results);

                vector<tuple<*double*, *double*, *double*>> data1, data2, data3;

                for (const *auto*& entry : results) {

                    data1.emplace\_back(get<0>(entry), get<1>(entry), get<3>(entry));

                    data2.emplace\_back(get<1>(entry), get<2>(entry), get<3>(entry));

                    data3.emplace\_back(get<2>(entry), get<0>(entry), get<3>(entry));

                    }

                write\_to\_file<tuple<*double*, *double*, *double*>>("3d\_data1.txt", data1);

                write\_to\_file<tuple<*double*, *double*, *double*>>("3d\_data2.txt", data2);

                write\_to\_file<tuple<*double*, *double*, *double*>>("3d\_data3.txt", data3);

            }

            break;

            case 2:{

                cout<<"Enter column quantity:"<<endl;

*int* n;

                cin>>n;

                cout<<"Enter column numbers:"<<endl;

                vector<*int*> columns;

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

                    string column;

                    cin>>column;

                    if (column.find("(") != string::npos) {

                        size\_t start = column.find("(") + 1;

                        size\_t end = column.find(")");

                        string val = column.substr(start, end - start);

                        columns.push\_back(stoi(val));

                        string col = column.substr(0, start - 1);

                        columns.push\_back(stoi(col));

                    } else {

                        columns.push\_back(stoi(column));

                    }

                }

                cout<<"Log or linear?"<<endl;

*bool* log;

                cin>>log;

                plot\_gnu(filename1,columns,columns\_title,log);

                cout.flush();

                cout<<endl;

                }

            break;

            case 3:{

                Population<1, 10, 5> pop = {};

*int* pop\_size;

*double* cros\_p, mut\_p;

                cout<<"Enter population size:"<<endl;

                cin>>pop\_size;

                cout<<"Enter crossingover probability:"<<endl;

                cout.flush();

                cin>>cros\_p;

                cout<<"Enter mutation probability:"<<endl;

                cin>>mut\_p;

                pop.fill(pop\_size);

                pop.setFunction([](*double* *x*) { return log(*x*)\*cos(3\**x*-15); });

                pop.setPCross(cros\_p);

                pop.setPMut(mut\_p);

                cout<<"Starting simulation..."<<endl;

                cout<<"Enter 0 to stop or n to do n steps: "<<endl;

*int* cont = 1;

                while (cont != 0)

                {

                    pop.doSteps(cont);

                    pop.draw();

                    cin>>cont;

                }

                cout<<"Simulation ended."<<endl;\

            }

            break;

            case 9:

                return -1;

            break;

            default:

                n=-1;

                continue;

        }

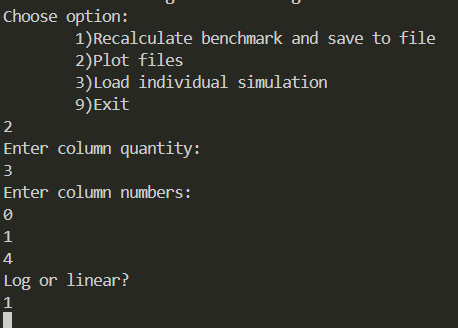
    }

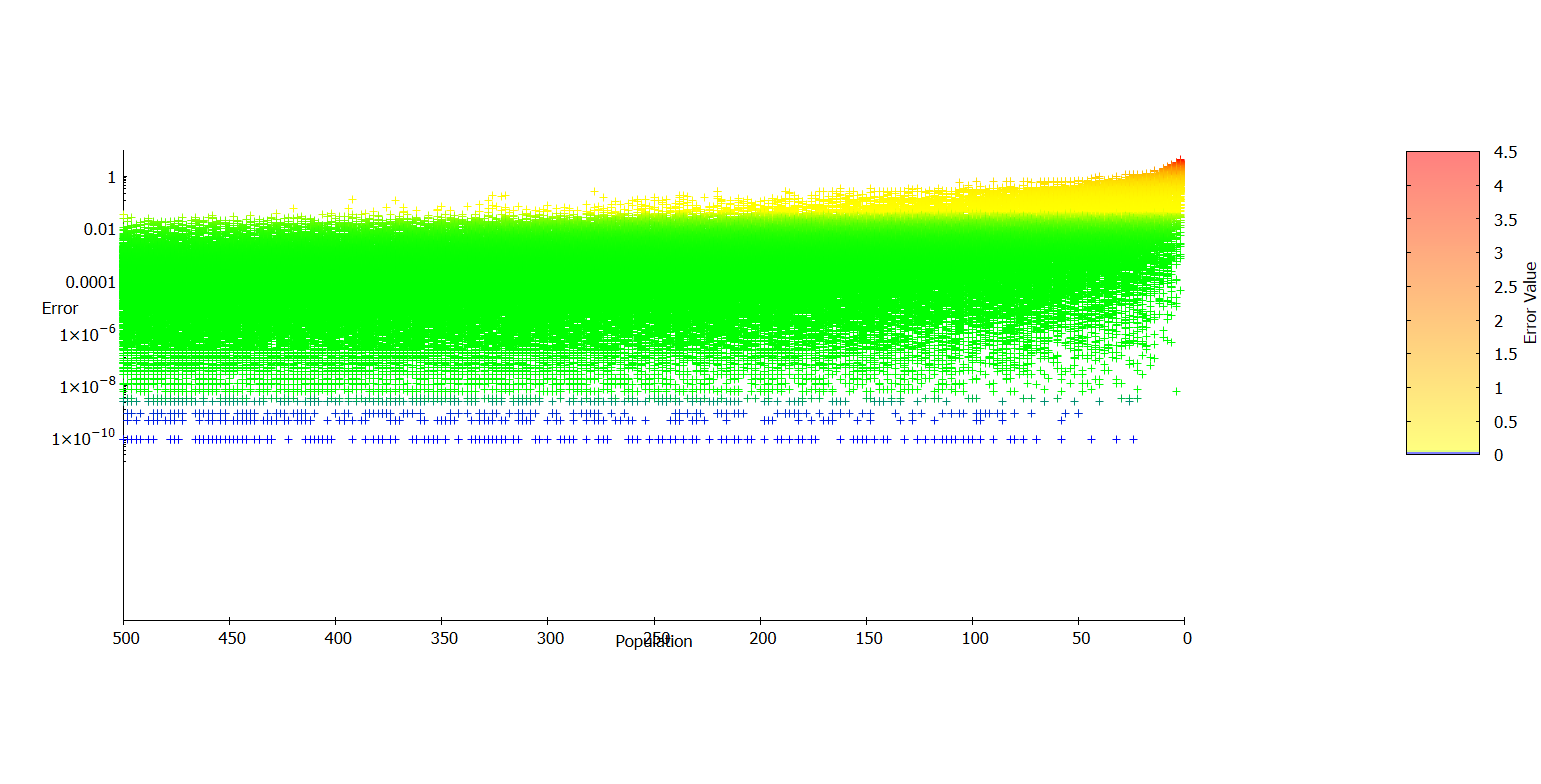
    return 0;

}

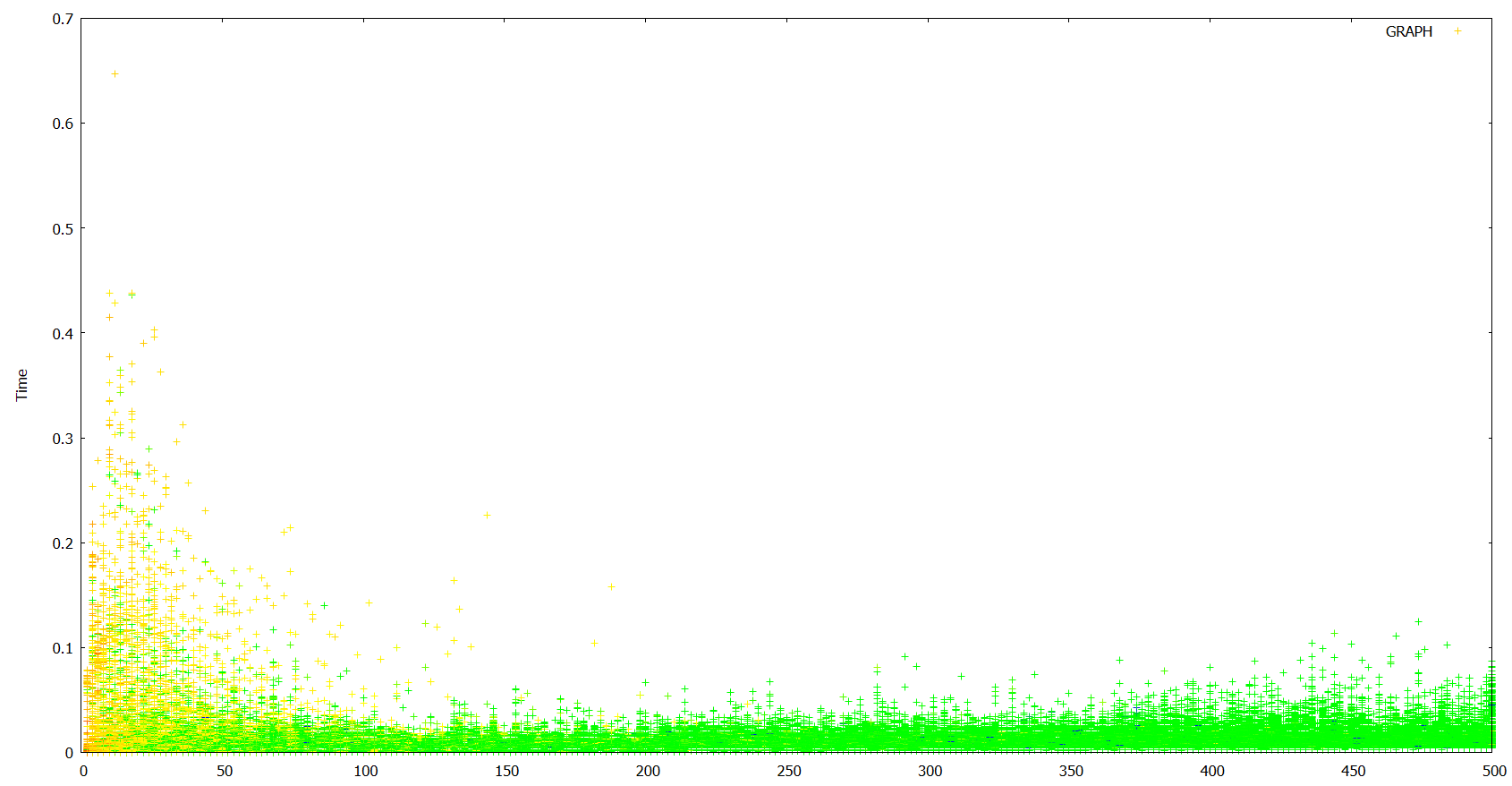
**4. Исследование**

С помощью функции **gen\_benchmark** я исследовал время выполнения и ошибку при различных наборах параметров.

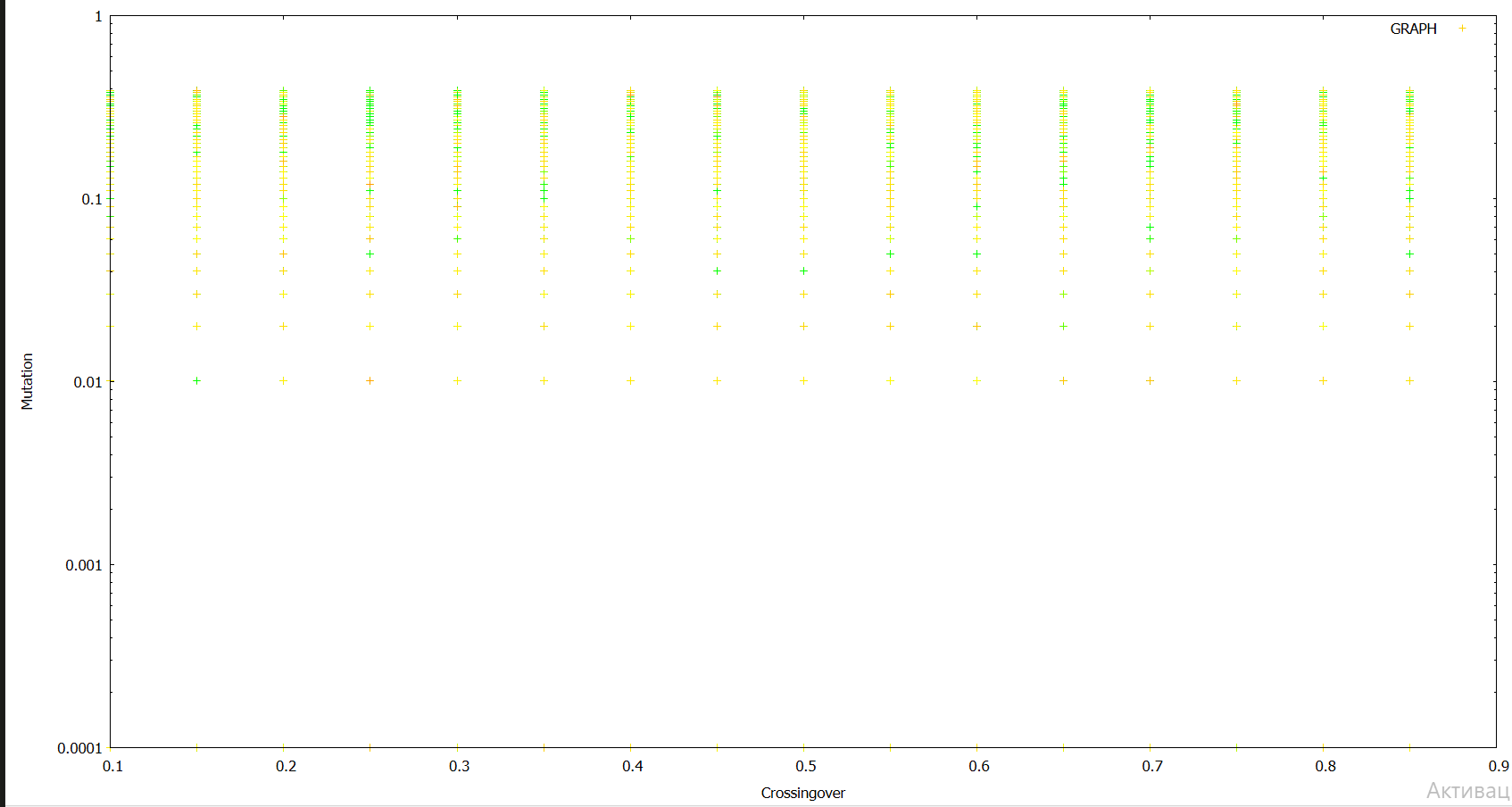
**

 График ошибки от мощности популяции

Заметно, что при малой популяции гораздо большая вероятность найти неверное решение.

  
График зависимости времени от мощности популяции (цвет – ошибка)

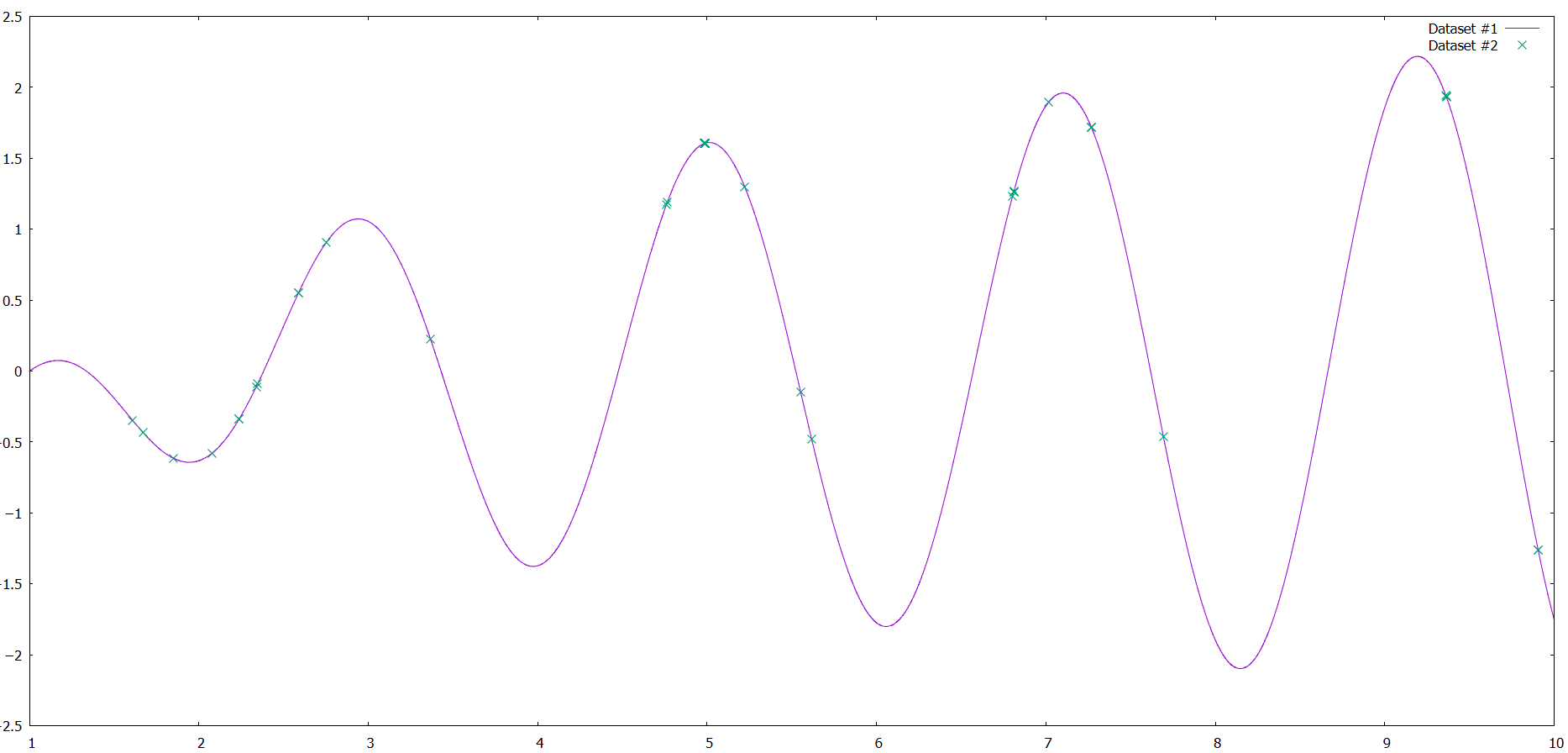
Также можно заметить что на малой популяции увеличивается время поиска ошибки. И хоть на совсем малых значения (<10) это время уменьшается, но растёт ошибка.

  
График зависимости ошибки от вероятностей мутации и кроссинговера.

Неожиданно, но при малой вероятности мутации (<0.1) высок уровень ошибки. Это модно объяснить тем, что популяция застревает в локальном экстремуме.

**5. Выполнение симуляции**

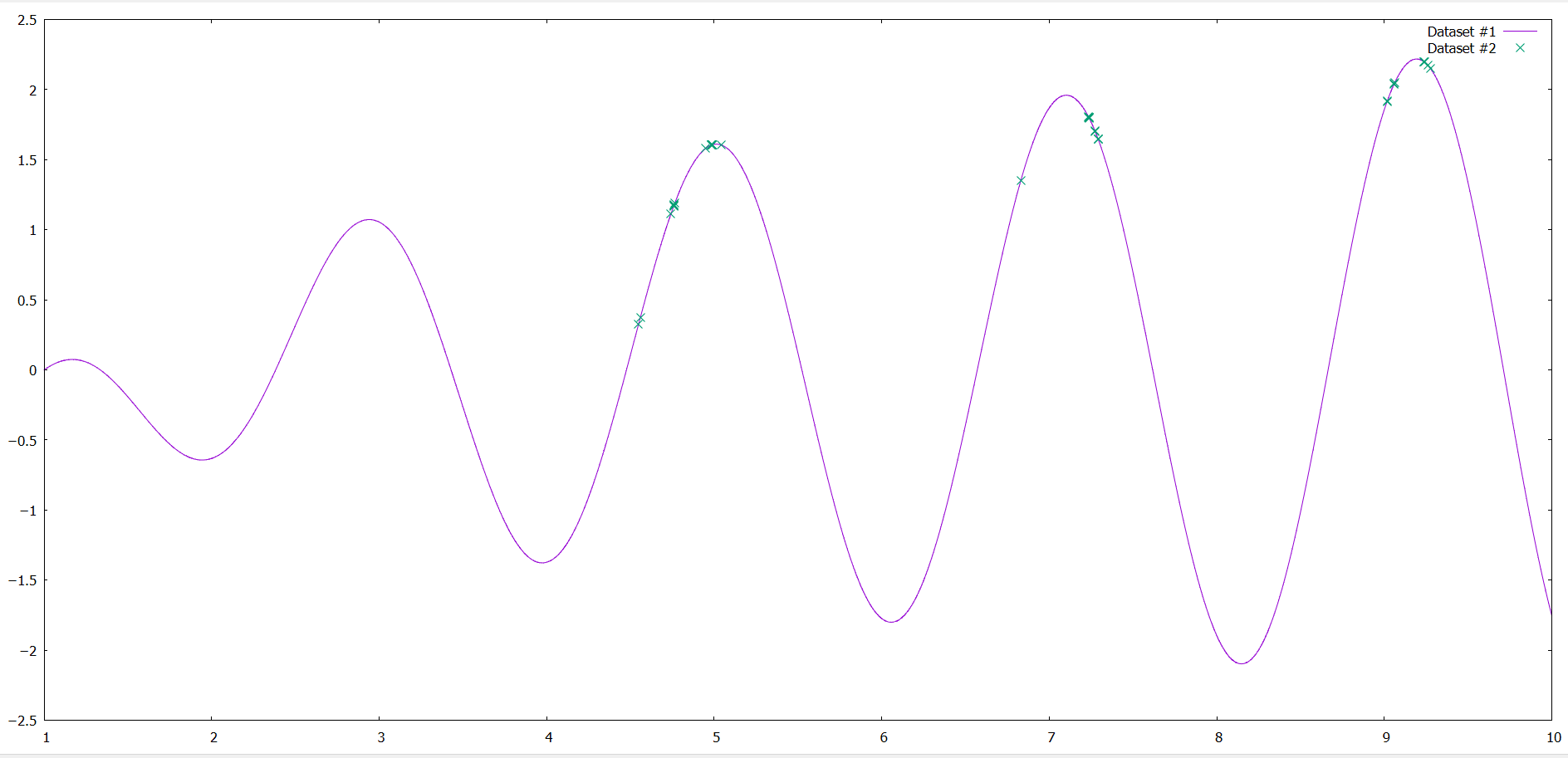
**Шаг 1**

****

Начальная популяция

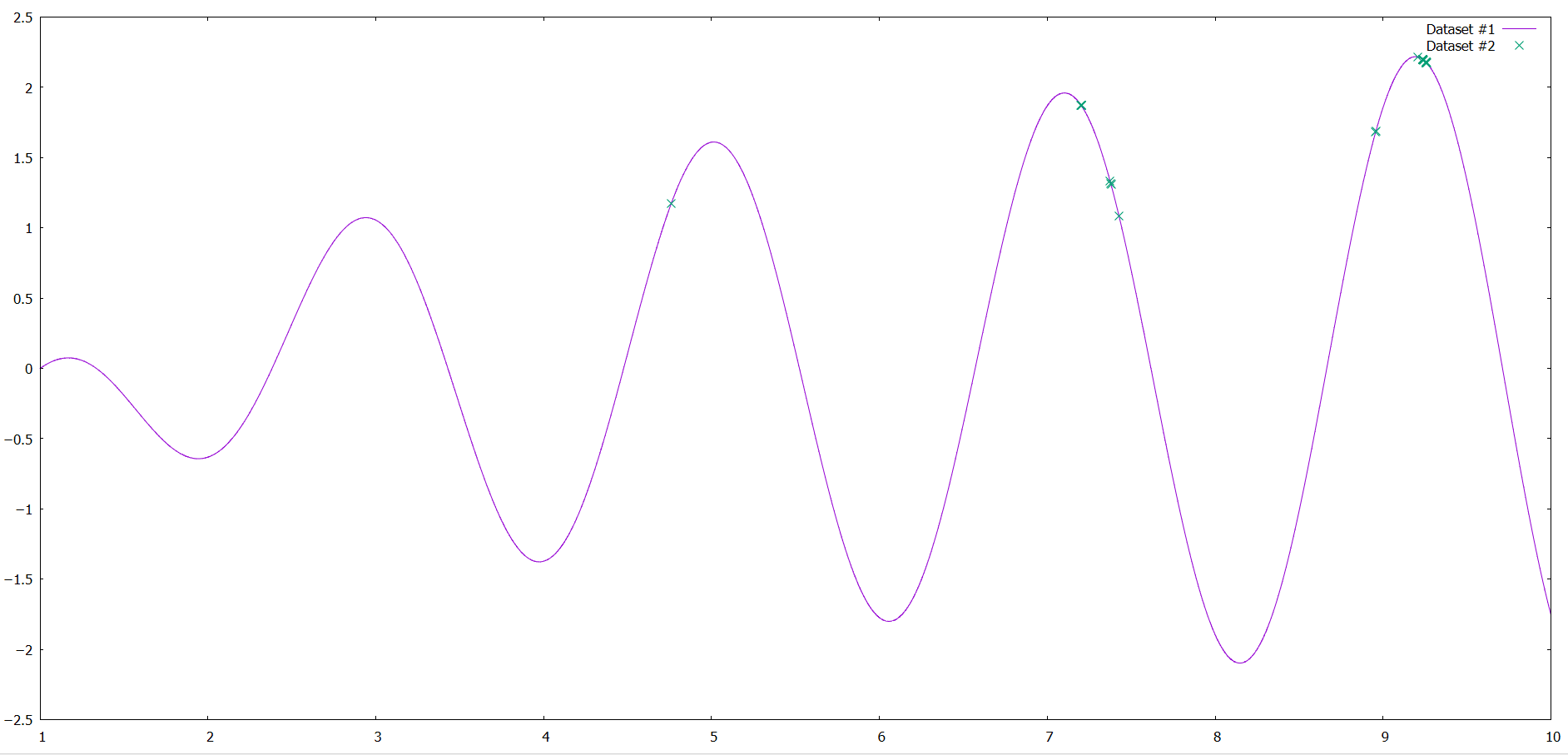
Лучшая точка (9.36202, 1.94136)

**Шаг 5**

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

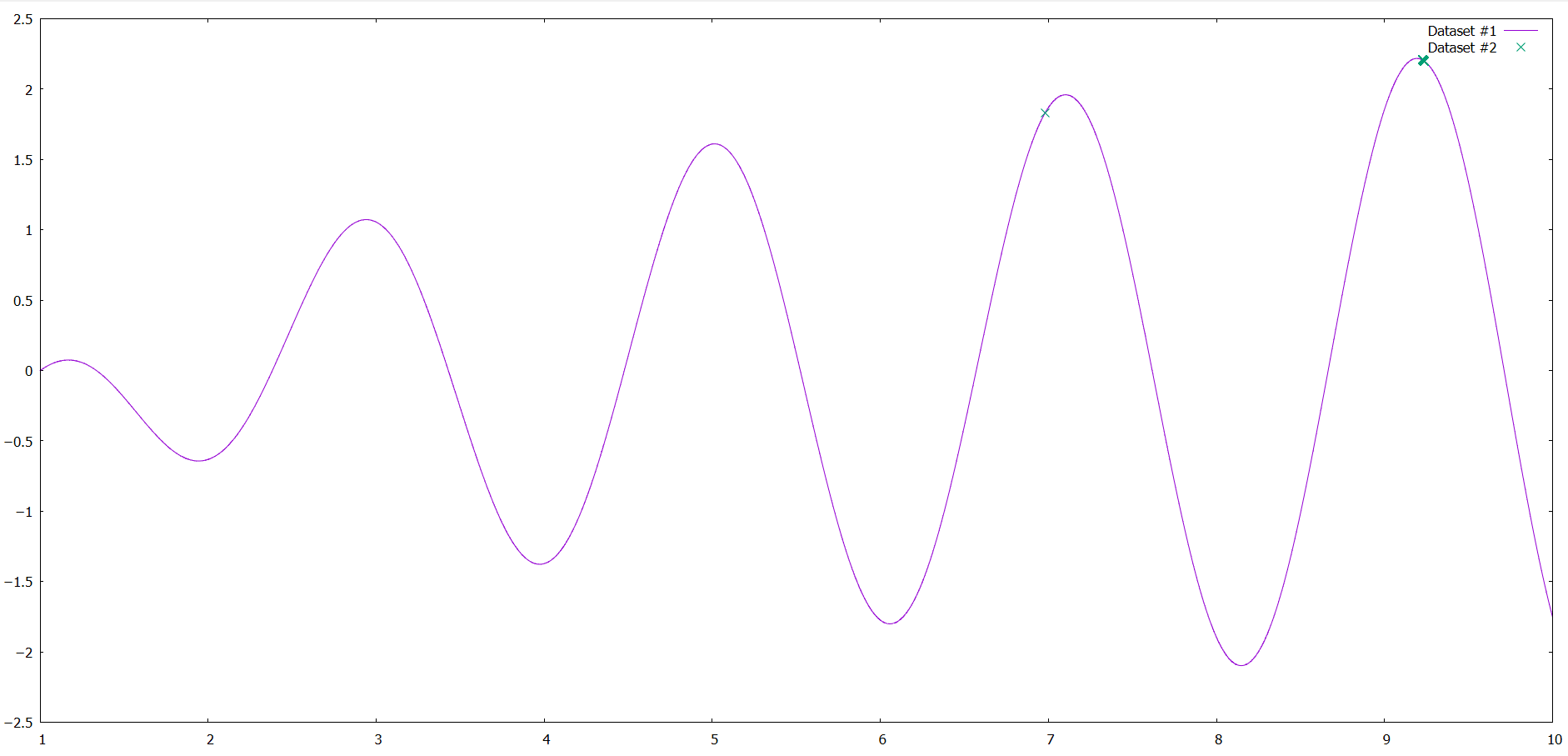
Лучшая точка (9.23649, 2.20044)

**Шаг 15**

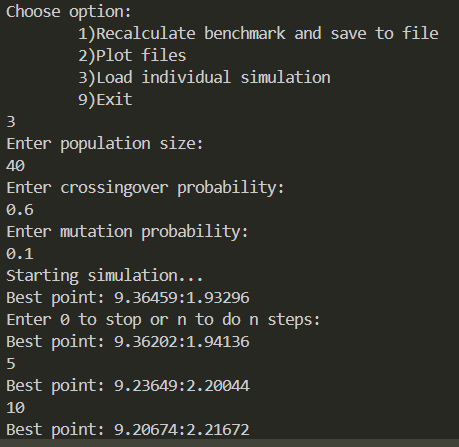
****Популяция спустя 15 шагов

Лучшая точка (9.20674, 2.21672)

**Шаг 45**

****Популяция спустя 45 шагов

Лучшая точка— (9.23641, 2.20051)

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

Точка экстремума — (9.19424, 2.21828)

Ошибка —(0.039, 0.018)

**6. Контрольный вопрос**

**Придумайте другую реализацию ОК.**

Вместо выбора места в хромосоме и обмена разрезанными частями, можно предложить другие варианты реализации кроссинговера.  
Например, хромосомы для каждого гена делается выбор – обменятся или нет, с вероятностью 50%.  
Пример: есть родители “1**23**4” и “7**89**0”, и случайно выпало, что меняются 2 и 3 гены, то тогда результатом кроссинговера будут “1894” и “7230”