|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | | |
|  | | | |
| Кафедра теоретической и прикладной информатики | | | |
|  | | | |
| Лабораторная работа № 4 | | | |
| по дисциплине «Методы оптимизации» | | | |
| **Статические методы поиска** | | | |
|  | | | |
|  | Бригада 2 | ПМ-81 Бортникова А.В., |
|  | ПМ-81 Ефремов А.А., |
|  | ПМ-81 Ртищева К.С. |
|
|
|
| Преподаватель | Чимитова Е.В. |
|  | | | |
| Новосибирск | | | |

1. **Цель работы**

Ознакомиться со статистическими методами поиска при реше­нии задач нелинейного программирования. Изучить методы случайного поиска при определении глобального экстремума функции.

1. **Задание**

Разработать программу для решения задачи поиска глобального экстремума с использованием метода простого случайного поиска и трех алгоритмов глобального поиска.

Исследовать метод простого случайного поиска глобального экстремума при различных  и .

Исследовать алгоритмы поиска глобального экстремума. Сравнить результаты поиска по количеству вычислений функции и найденной точке экстремума. Исследование провести при различных значениях числа попыток . Повторить при 5 разных начальных значениях ГСЧ. Сделать выводы об устойчивости различных алгоритмов.

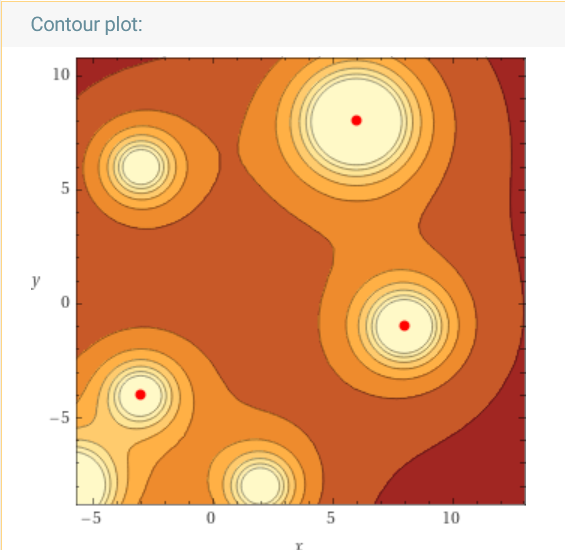
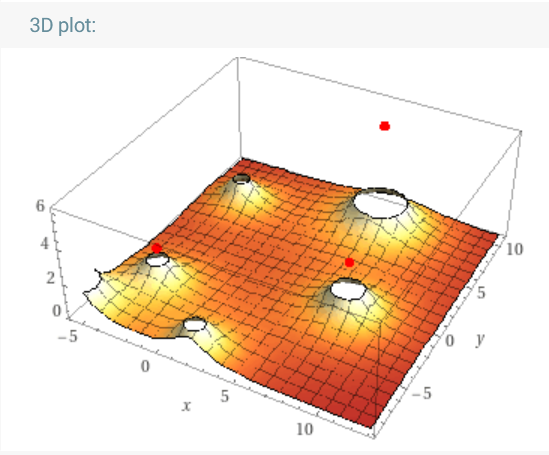
*Вариант 2:*

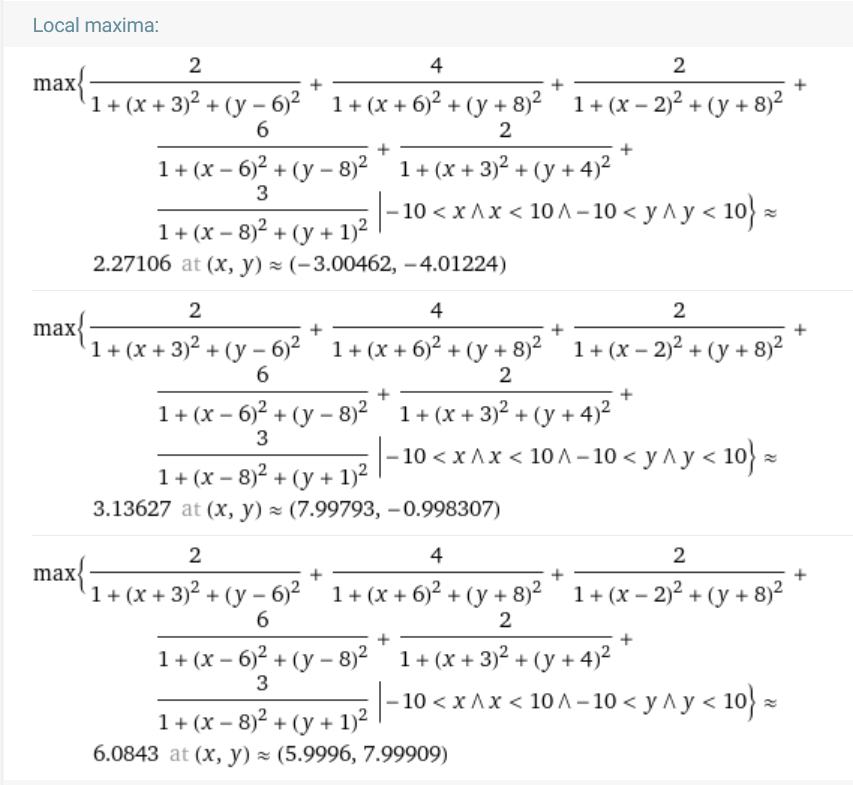
Найти максимум заданной функции:



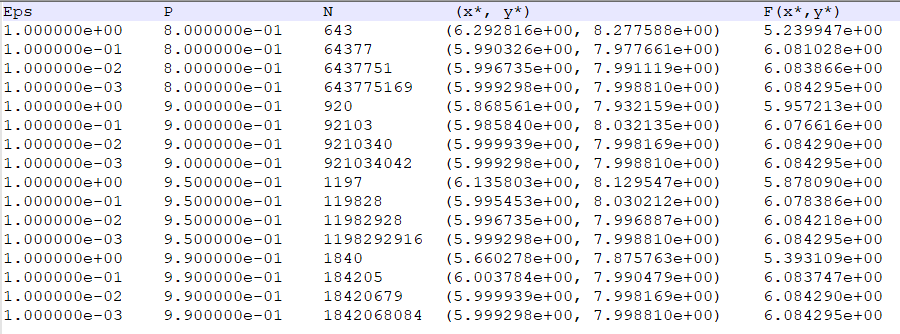
на области , .

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **№** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **2** | 2 | 4 | 2 | 6 | 2 | 3 | -3 | -6 | 2 | 6 | -3 | 8 | 6 | -8 | -8 | 8 | -4 | -1 |



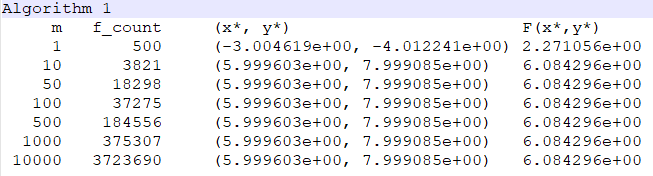


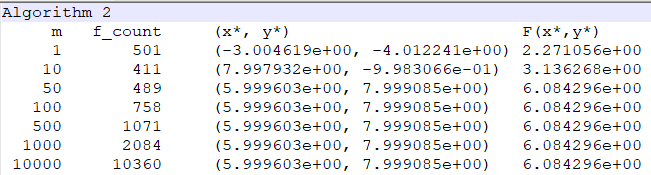
1. **Исследование метода простого случайного поиска глобального экстремума при различных ξ и Р.**

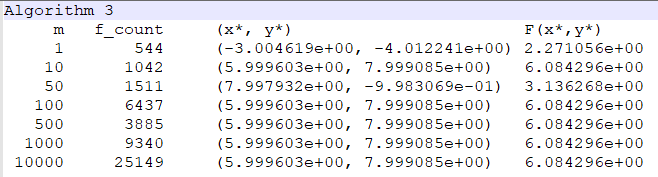


При увеличении Р - вероятности того, что мы найдем решение за N испытаний, и при увеличении требуемой точности решения ξ увеличивается число итераций N и, соответственно, число вычислений функции.

1. **Исследование алгоритмов поиска глобального экстремума**



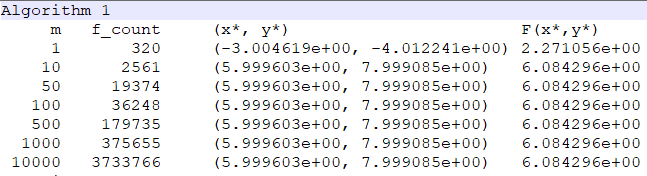


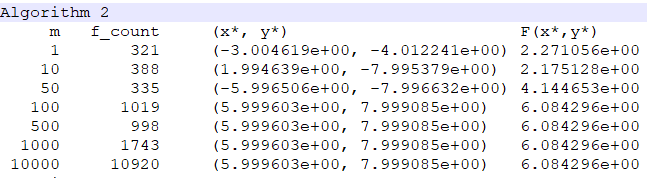


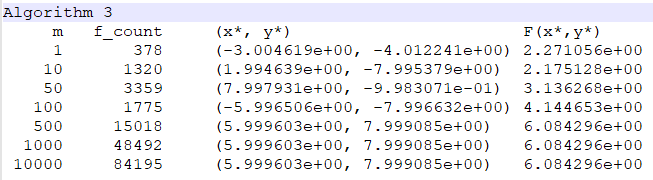
В результате сравнения алгоритмов при различных значениях числа попыток m можно сделать вывод о том, что при маленьком значении m методы сходятся не в глобальный экстремум, а в локальный. Также видно, что второй алгоритм эффективнее остальных, т.к. он имеет меньшее число вычислений функции.

1. **Исследование алгоритмов поиска глобального экстремума при 5 различных начальных значениях ГСЧ**

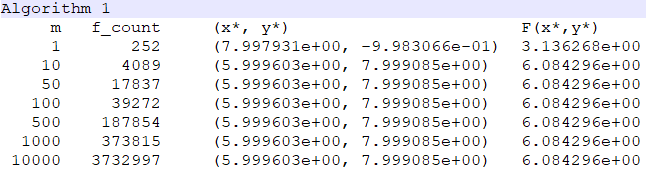
**num = 10**

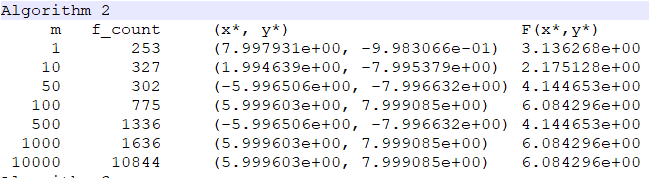


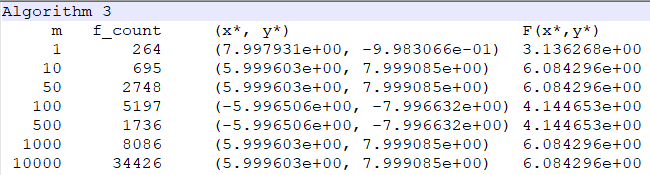




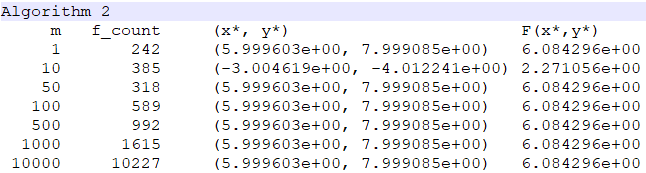
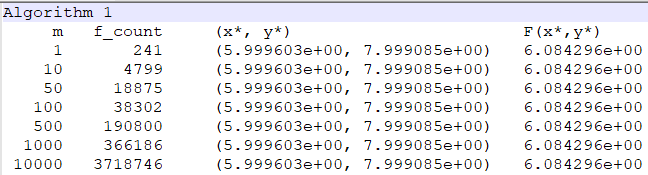
**num = 35234**

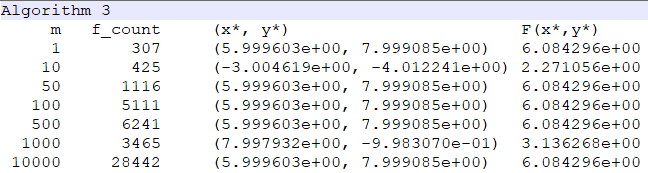




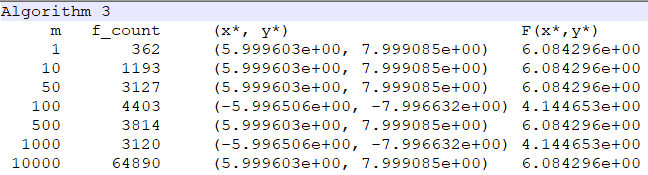
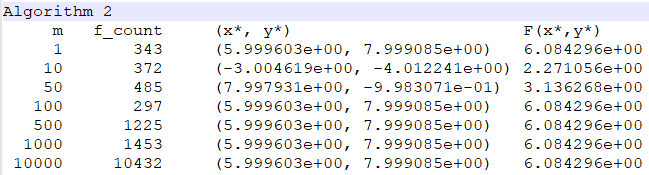
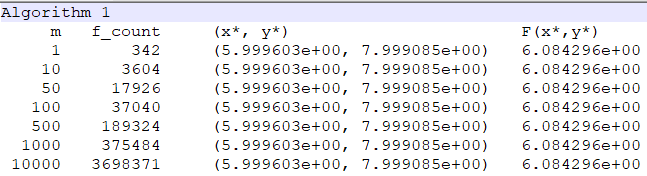


**num = -234**

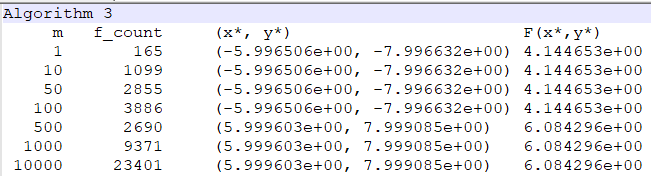
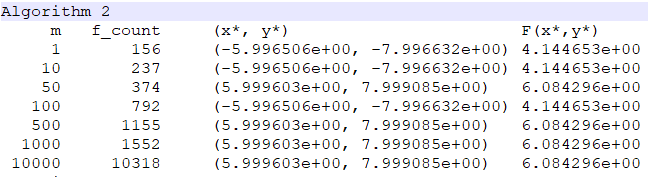
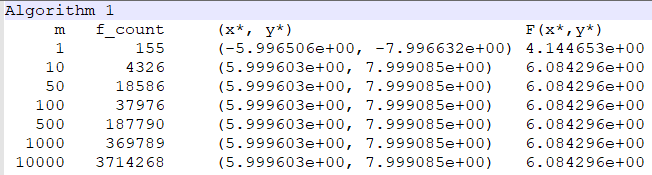




**num = -98765**



**num = 4957592**



По результатам данного исследования можно сделать вывод о том, что первый алгоритм устойчив к выбору различного начального значения ГСЧ, что нельзя сказать о двух других.

1. **Программа**

*Файл «OptMethLab4.cpp»*

#include <fstream>

#include <iostream>

#include <cstdlib>

#include <random>

#include <vector>

#include "DanPshen.h"

using namespace std;

void SimpleRandomSearch(ofstream& fout, double p, double eps, vector<double>& x, vector<double>& y)

{

double p\_e, Func;

double Fmin = numeric\_limits<double>::infinity();

vector<double> minPoint(2), point(2);

p\_e = eps \* eps / ((y[1] - y[0]) \* (x[1] - x[0]));

int N = ceil(log(1 - p) / log(1 - p\_e));

for (int i = 0; i < N; i++)

{

static const double fraction = 1.0 / (static\_cast<double>(RAND\_MAX) + 1.0);

point[0] = static\_cast<double>(rand() \* fraction \* (x[1] - x[0] + 1) + x[0]);

point[1] = static\_cast<double>(rand() \* fraction \* (y[1] - y[0] + 1) + y[0]);

Func = F(point);

if (Func < Fmin)

{

Fmin = Func;

minPoint = point;

}

}

fout << scientific << eps << "\t" << p << "\t" << N << "\t (" << minPoint[0] << ", " << minPoint[1] << ")\t" << -Fmin << endl;

}

void SimpleRandomSearchTest(vector<double>& p, vector<double>& eps, vector<double>& x, vector<double>& y)

{

ofstream fout;

fout.open("SimpleRandomSearch.txt");

fout << "Eps P N (x\*, y\*) F(x\*,y\*)" << endl;

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

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

SimpleRandomSearch(fout, p[i], eps[j], x, y);

fout.close();

}

void GlobalSearch\_1(double eps, vector<double>& x, vector<double>& y, vector<int>& m, ofstream& fout, int& num)

{

mt19937 gen(num);

uniform\_real\_distribution<> distr(x[0], x[1]);

DanPshen dp(2);

fout << "Algorithm 1" << endl;

fout << " m f\_count (x\*, y\*) F(x\*,y\*)" << endl;

vector<double> minPoint(2), point(2);

for (int i = 0; i < m.size(); i++)

{

double Fmin = numeric\_limits<double>::infinity(), Func;

int f\_calc\_count = 0;

for (int j = 0; j < m[i]; j++)

{

/\*static const double fraction = 1.0 / (static\_cast<double>(RAND\_MAX) + 1.0);

point[0] = static\_cast<double>(rand() \* fraction \* (x[1] - x[0] + 1) + x[0]);

point[1] = static\_cast<double>(rand() \* fraction \* (y[1] - y[0] + 1) + y[0]);\*/

point[0] = distr(gen);

point[1] = distr(gen);

dp.FindExtremum(F, FindMinArgGolden, point, eps, eps);

Func = F(dp.xk1);

f\_calc\_count++;

if (Func < Fmin)

{

Fmin = Func;

minPoint = dp.xk1;

}

f\_calc\_count += dp.f\_calc\_count;

}

fout << setw(6) << m[i] << "\t" << setw(8) << f\_calc\_count << scientific << "\t (" << minPoint[0] << ", " << minPoint[1];

fout << ")\t" << -Fmin << endl;

}

}

void GlobalSearch\_2(double eps, vector<double>& x, vector<double>& y, vector<int>& m, ofstream& fout, int& num)

{

mt19937 gen(num);

uniform\_real\_distribution<> distr(x[0], x[1]);

DanPshen dp(2);

fout << "Algorithm 2" << endl;

fout << " m f\_count (x\*, y\*) F(x\*,y\*)" << endl;

vector<double> minPoint(2), point(2);

double Fmin, Func;

for (int j = 0; j < m.size(); j++)

{

/\*static const double fraction = 1.0 / (static\_cast<double>(RAND\_MAX) + 1.0);

point[0] = static\_cast<double>(rand() \* fraction \* (x[1] - x[0] + 1) + x[0]);

point[1] = static\_cast<double>(rand() \* fraction \* (y[1] - y[0] + 1) + y[0]);\*/

point[0] = distr(gen);

point[1] = distr(gen);

dp.FindExtremum(F, FindMinArgGolden, point, eps, eps);

minPoint = dp.xk1;

Fmin = F(dp.xk1);

int f\_calc\_count = dp.f\_calc\_count + 1;

int i;

while (true)

{

for (i = 0; i < m[j]; i++)

{

/\*static const double fraction = 1.0 / (static\_cast<double>(RAND\_MAX) + 1.0);

point[0] = static\_cast<double>(rand() \* fraction \* (x[1] - x[0] + 1) + x[0]);

point[1] = static\_cast<double>(rand() \* fraction \* (y[1] - y[0] + 1) + y[0]);\*/

point[0] = distr(gen);

point[1] = distr(gen);

Func = F(point);

f\_calc\_count++;

if (Func < Fmin)

{

Fmin = Func;

minPoint = point;

break;

}

}

if (i == m[j]) { break; }

dp.FindExtremum(F, FindMinArgGolden, minPoint, eps, eps);

minPoint = dp.xk1;

Fmin = F(dp.xk1);

f\_calc\_count += dp.f\_calc\_count + 1;

}

fout << setw(6) << i << "\t" << setw(8) << f\_calc\_count << scientific << "\t (" << minPoint[0] << ", " << minPoint[1] << ")\t";

fout << -Fmin << endl;

}

}

void GlobalSearch\_3(double eps, vector<double>& x, vector<double>& y, vector<int>& m, ofstream& fout, int& num)

{

mt19937 gen(num);

uniform\_real\_distribution<> distr(x[0], x[1]);

DanPshen dp(2);

fout << "Algorithm 3" << endl;

fout << " m f\_count (x\*, y\*) F(x\*,y\*)" << endl;

vector<double> minPoint(2), point(2), direction(2);

double Fmin, Func;

for (int j = 0; j < m.size(); j++)

{

/\*static const double fraction = 1.0 / (static\_cast<double>(RAND\_MAX) + 1.0);

point[0] = static\_cast<double>(rand() \* fraction \* (x[1] - x[0] + 1) + x[0]);

point[1] = static\_cast<double>(rand() \* fraction \* (y[1] - y[0] + 1) + y[0]);\*/

point[0] = distr(gen);

point[1] = distr(gen);

dp.FindExtremum(F, FindMinArgGolden, point, eps, eps);

minPoint = dp.xk1;

point = minPoint;

Fmin = F(dp.xk1);

int i, f\_calc\_count = dp.f\_calc\_count + 1;

double delta = 0.5;

while (true)

{

for (i = 0; i < m[j]; i++)

{

/\*static const double fraction = 1.0 / (static\_cast<double>(RAND\_MAX) + 1.0);

direction[0] = static\_cast<double>(rand() \* fraction \* (x[1] - x[0] + 1) + x[0]);

direction[1] = static\_cast<double>(rand() \* fraction \* (y[1] - y[0] + 1) + y[0]);\*/

direction[0] = distr(gen);

direction[1] = distr(gen);

vector<double> precPoint = point;

do

{

point[0] += delta \* (direction[0] / sqrt(direction[0] \* direction[0] + direction[1] \* direction[1]));

point[1] += delta \* (direction[1] / sqrt(direction[0] \* direction[0] + direction[1] \* direction[1]));

f\_calc\_count += 2;

} while (point[0] > -10 && point[0] <= 10 && point[1] > -10 && point[1] <= 10 && F(precPoint) <= F(point));

if (point[0] > -10 && point[0] <= 10 && point[1] > -10 && point[1] <= 10) { break; }

}

if (i == m[j]) { break; }

dp.FindExtremum(F, FindMinArgGolden, point, eps, eps);

f\_calc\_count += dp.f\_calc\_count + 1;

Func = F(dp.xk1);

if (Func < Fmin)

{

Fmin = Func;

minPoint = dp.xk1;

}

point = minPoint;

}

fout << setw(6) << i << "\t" << setw(8) << f\_calc\_count << scientific << "\t (" << minPoint[0] << ", " << minPoint[1] << ")\t";

fout << -Fmin << endl;

}

}

int main()

{

vector<double> x = { -10, 10 };

vector<double> y = { -10, 10 };

vector<double> p = { 0.8, 0.9, 0.95, 0.99 };

vector<double> eps = { 1, 0.1, 0.01, 0.001 };

vector<int> m = { 1, 10, 50, 100, 500, 1000, 10000 };

int num = 4957592;

SimpleRandomSearchTest(p, eps, x, y);

ofstream fout;

fout.open("GlobalSearc.txt");

GlobalSearch\_1(1e-6, x, y, m, fout, num);

GlobalSearch\_2(1e-6, x, y, m, fout, num);

GlobalSearch\_3(1e-6, x, y, m, fout, num);

fout.close();

}

*Файл «DanPshen.h»*

#pragma once

#include <vector>

#include "Lab2Data.h"

#include "Vector.h"

class DanPshen

{

public:

const int size; // Размерность вектора

vector<double> Sk; // Минус гардиент на текущем шаге

vector<double> xk; // Приближение на текущем шаге

vector<double> xk1; // Новое приближение

vector<double> grad\_xk1; // Вспомогательный вектор для нахождения экстремума

vector<double> t; // Вспомогательный вектор для нахождения градиента

int f\_calc\_count = 0; // Число вычислений функции

DanPshen() : size(0)

{

}

DanPshen(const int& t\_size) : size(t\_size)

{

Sk.resize(size);

xk.resize(size);

xk1.resize(size);

grad\_xk1.resize(size);

t.resize(size);

}

int FindExtremum(double funct(const vector<double>&),

int min\_max(double f(const vector<double>&), const vector<double>&, const vector<double>&, const double&, double&),

const vector<double>& x0, const double& f\_eps, const double& xs\_eps)

{

f\_calc\_count = 0;

// 1. Расчет градиента функции f в точке x0

Grad(x0, Sk);

f\_calc\_count += 4;

Sk \*= -1;

xk = x0;

bool exit\_flag = true;

int iter\_count = 0;

do

{

// 2. Минимизация функции f по направлению Sk

//Function to\_minimize = Function(xk, Sk);

double lambda;

f\_calc\_count += min\_max(funct, xk, Sk, 1e-5, lambda) + 1;

// Получение нового приближения

xk1 = xk + lambda \* Sk;

// 3. Вычисление grad(f(xk1)) и весового коэффициента omega

Grad(xk1, grad\_xk1);

f\_calc\_count += 4;

//double omega = (grad\_xk1 \* (grad\_xk1 + Sk)) / (-1 \* (Sk \* Sk));

double omega = (grad\_xk1 \* grad\_xk1) / (Sk \* Sk);

// 4. Определение новго направления Sk1

Sk = -1 \* grad\_xk1 + omega \* Sk;

// Расчет изменения решения на текущей итерации

exit\_flag = true;

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

if(abs(xk[i] - xk1[i]) > xs\_eps)

exit\_flag = false;

xk = xk1;

iter\_count++;

} while(Norm(Sk) > f\_eps && iter\_count < 100 && exit\_flag == false);

return iter\_count;

}

};

*Файл «Lab2Data.h»*

#pragma once

#include <vector>

#include "Vector.h"

using namespace std;

const double SQRT5 = sqrt(5);

const double PI = 3.14159265;

double F(const vector<double>& point)

{

double sum = 0.;

vector<int> c = { 2, 4, 2, 6, 2, 3 };

vector<int> a = { -3, -6, 2, 6, -3, 8 };

vector<int> b = { 6, -8, -8, 8, -4, -1 };

for (size\_t i = 0; i < 6; i++)

sum += c[i] / (1.0 + (point[0] - a[i]) \* (point[0] - a[i]) + (point[1] - b[i]) \* (point[1] - b[i]));

return -sum;

}

void Grad(const vector<double>& point, vector<double>& res)

{

vector<int> c = { 2, 4, 2, 6, 2, 3 };

vector<int> a = { -3, -6, 2, 6, -3, 8 };

vector<int> b = { 6, -8, -8, 8, -4, -1 };

double sum\_x = 0., sum\_y = 0.;

for (size\_t i = 0; i < 6; i++)

{

sum\_x -= 2 \* c[i] \* (point[0] - a[i]) / ((1.0 + (point[0] - a[i]) \* (point[0] - a[i]) + (point[1] - b[i]) \* (point[1] - b[i])) \*

(1.0 + (point[0] - a[i]) \* (point[0] - a[i]) + (point[1] - b[i]) \* (point[1] - b[i])));

sum\_y -= 2 \* c[i] \* (point[1] - b[i]) / ((1.0 + (point[0] - a[i]) \* (point[0] - a[i]) + (point[1] - b[i]) \* (point[1] - b[i])) \*

(1.0 + (point[0] - a[i]) \* (point[0] - a[i]) + (point[1] - b[i]) \* (point[1] - b[i])));

}

res[0] = sum\_x;

res[1] = sum\_y;

}

// Поиск отрезка с минимумом функции

int FindSegmentWithMin(const double& delta, double funct(const vector<double>&),

const vector<double>& x, const vector<double>& Sk, double& a, double& b)

{

double x0 = 0;

double xk, xk1, xk\_1, h = 2;

double f = funct(x + (x0) \* Sk);

int f\_calc\_count = 1;

if(f == funct(x + (x0 + delta) \* Sk))

{

a = x0;

b = x0 + delta;

return 2;

}

else if(f == funct(x + (x0 - delta) \* Sk))

{

a = x0 - delta;

b = x0;

return 3;

}

else

{

if(f > funct(x + (x0 + delta) \* Sk))

{

xk = x0 + delta;

h = delta;

f\_calc\_count++;

}

else if(f > funct(x + (x0 - delta) \* Sk))

{

xk = x0 - delta;

h = -delta;

f\_calc\_count += 2;

}

else

{

a = x0 - delta;

b = x0 + delta;

return f\_calc\_count + 2;

}

xk\_1 = x0;

bool exit = false;

do

{

h \*= 2;

xk1 = xk + h;

if(funct(x + (xk)\*Sk) > funct(x + (xk1)\*Sk))

{

xk\_1 = xk;

xk = xk1;

}

else

exit = true;

f\_calc\_count += 2;

} while(!exit);

a = xk\_1;

b = xk;

}

return f\_calc\_count;

}

// Поиск аргумента минимума функции вдоль направления методом золотого сечения

int FindMinArgGolden(double funct(const vector<double>&),

const vector<double>& x, const vector<double>& Sk, const double& eps, double& result)

{

double a = 0, b = 0;

int f\_calc\_count = FindSegmentWithMin(1e-1, funct, x, Sk, a, b);

double x1 = a + (3 - SQRT5) / 2 \* (b - a);

double x2 = a + (SQRT5 - 1) / 2 \* (b - a);

double f1 = funct(x + (x1)\*Sk);

double f2 = funct(x + (x2)\*Sk);

double a1, b1;

int iter\_count = 0;

for(; abs(b - a) > eps; iter\_count++)

{

a1 = a, b1 = b;

if(f1 < f2)

{

b = x2;

x2 = x1;

x1 = a + (3 - SQRT5) / 2 \* (b - a);

f2 = f1;

f1 = funct(x + (x1)\*Sk);

}

else

{

a = x1;

x1 = x2;

x2 = a + (SQRT5 - 1) / 2 \* (b - a);

f1 = f2;

f2 = funct(x + (x2)\*Sk);

}

}

result = a;

return iter\_count + f\_calc\_count;

}

*Файл «Vector.h»*

#pragma once

#include <vector>

#include <iomanip>

#include <fstream>

using namespace std;

// Умножение вектора на число

vector<double> operator \* (const double& val, vector<double> vec)

{

const size\_t size = vec.size();

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

vec[i] \*= val;

return vec;

}

// Деление вектора на число

vector<double> operator / (const double& val, vector<double> vec)

{

const size\_t size = vec.size();

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

vec[i] /= val;

return vec;

}

vector<double>& operator \*= (vector<double>& vec, const double& val)

{

const size\_t size = vec.size();

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

vec[i] \*= val;

return vec;

}

// Сложение векторов

vector<double> operator + (vector<double> vec1, const vector<double>& vec2)

{

const size\_t size = vec1.size();

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

vec1[i] += vec2[i];

return vec1;

}

// Вычитание векторов

vector<double> operator - (vector<double>vec1, const vector<double>& vec2)

{

const size\_t size = vec1.size();

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

vec1[i] -= vec2[i];

return vec1;

}

// Скалярное произведение векторов

double operator \* (const vector<double>& vec1, const vector<double>& vec2)

{

const size\_t size = vec1.size();

double res = 0;

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

res += vec1[i] \* vec2[i];

return res;

}

// Норма вектора

double Norm(const vector<double>& vec)

{

const size\_t size = vec.size();

double res = 0;

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

res += vec[i] \* vec[i];

return sqrt(res);

}