Министерство науки и высшего образования Российской Федерации федеральное государственное автономное образовательное учреждение высшего образования **«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»**

Отчет

по **Лабораторной работе №2**

по дисциплине **«Методы Оптимизации»**

Работа учеников

группы М32351  
Юльцовой Натальи,  
Колчина Дмитрия

2021

**Цель и задачи работы:**

Цель : исследовать методы решения задачи минимизации квадратичной функции.

Задачи:

1. Реализовать алгоритмы:

* метод градиентного спуска;
* метод наискорейшего спуска;
* метод сопряженных градиентов.

Оценить, как меняется скорость сходимости, если для поиска величины шага использовать различные методы одномерного поиска.

2. Проанализировать траектории методов для нескольких квадратичных функций.

3. Исследовать, как зависит число итераций, необходимое методам для сходимости, от следующих двух параметров: а) числа обусловленности 𝑘 ≥ 1 оптимизируемой функции; б) размерности пространства 𝑛 оптимизируемых переменных.

Ход работы.

1. Возьмем размерности(n) 10, 100, 1000 и числа обусловленности(k) от 20 до 1820 с шагом 200. Рассмотрим различные методы одномерного поиска для метода наискорейшего спуска и зависимость от этого числа итераций, а также сравним с другими градиентными методами:

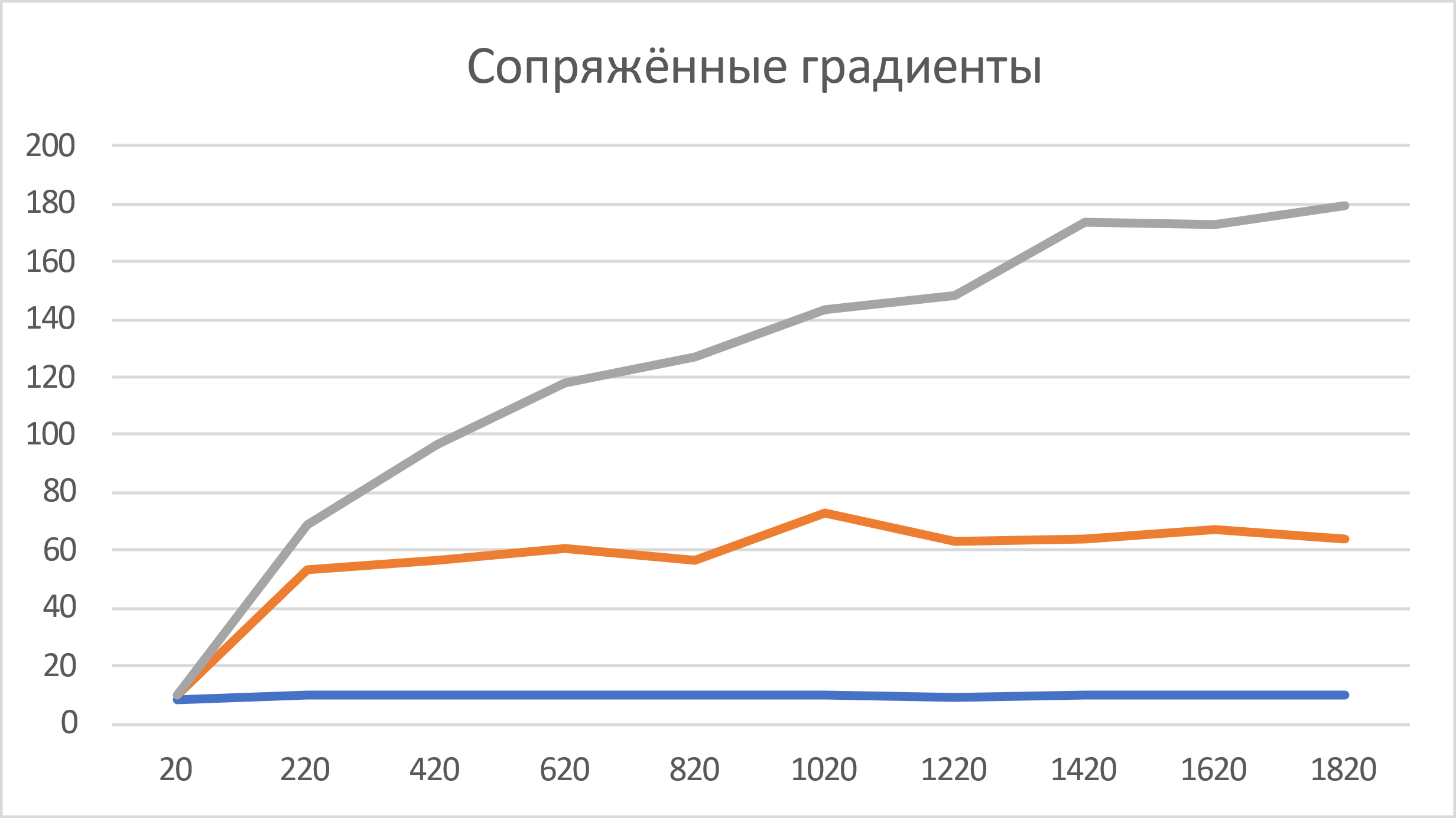
*Голубая линия – размерность 10*

*Оранжевая – 100*

*Серая - 1000*

Как можно видеть количество итераций одинаково. Значит, не важно какой метод одномерной сходимости выбирать – это не влияет на количество итераций. Это происходит потому, что шаг альфа не меняется. Так же количество итераций не зависит от размерности пространства, увеличивается в зависимости от числа обусловленности.

Метод градиентного спуска делает наибольшее количество итераций и от увеличения размерности почти не меняется, а при увеличении числа обусловленности возрастает пропорционально.



Количество итераций не превышает размерности пространства в методе сопряженных градиентов.

Сходимость каждого метода линейна, так как при выводе альфа на каждой итерации практически не меняется.

На следующем графике можем заметить, что метод сопряженных градиентов справляется за наименьшее число итераций.

Голубая линия – метод сопряженных градиентов

Серая – метод наискорейшего спуска

Оранжевая – метод градиентного спуска

2. Проанализируем траектории методов для следующих квадратичных функций: (обозначены желтым цветом)

1. f(x,y) = 64x2+126xy+64y2-10x+30y+13

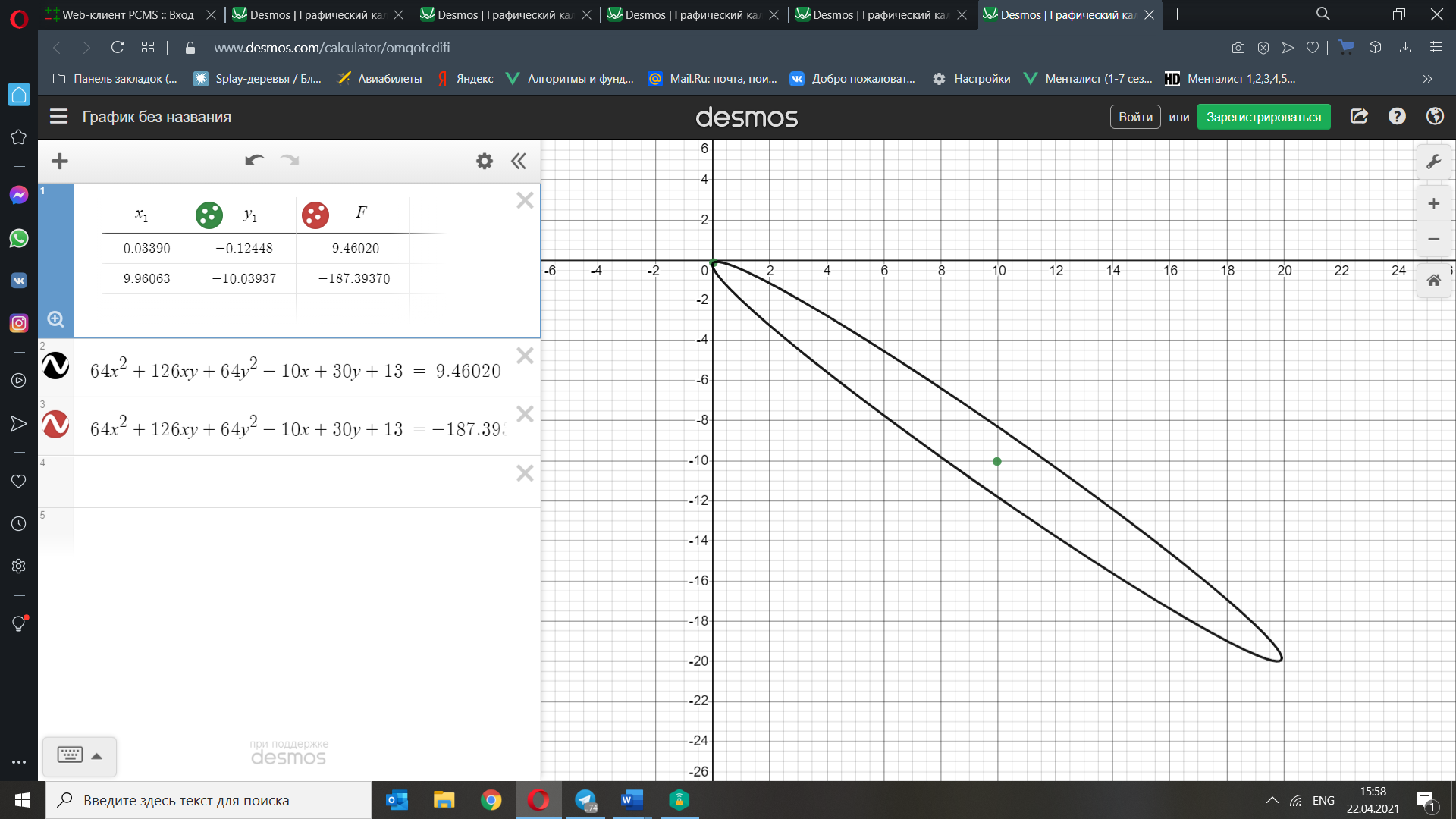
* Аналитическое решение:

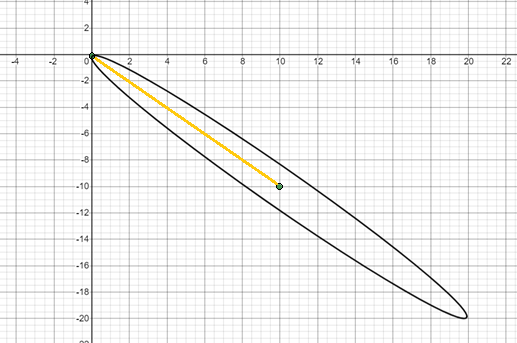
Найдем частные производные функции и приравняем к нулю. Получаем минимум в точке

x= 9,96063 y = -10,03937

* метод сопряженных градиентов.

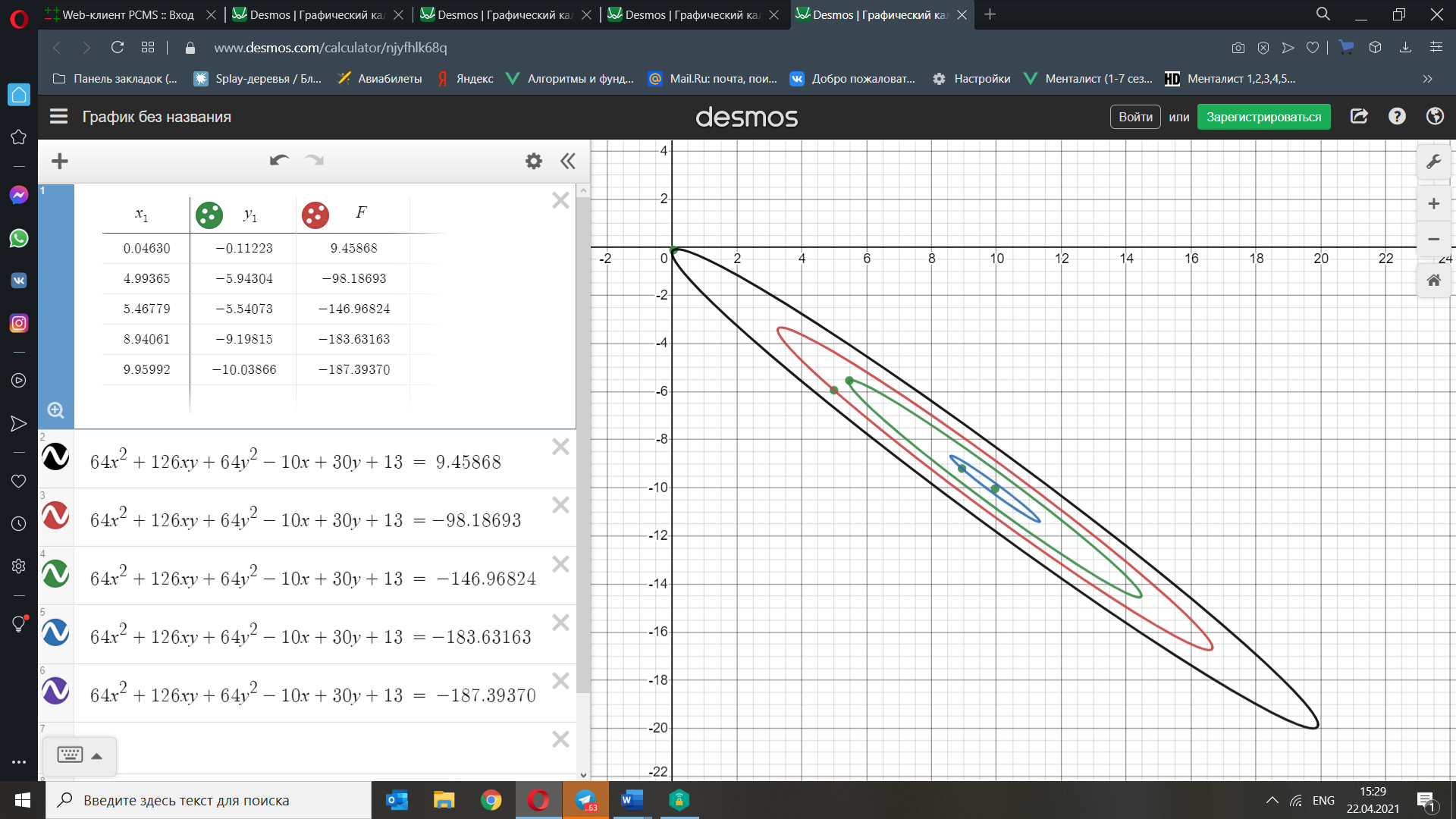
(2 итерации)

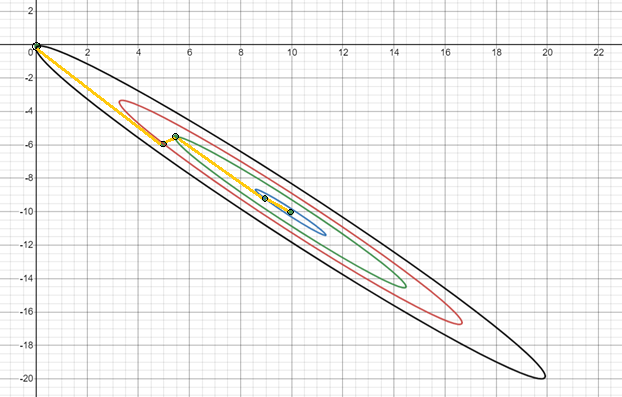




* метод наискорейшего спуска;

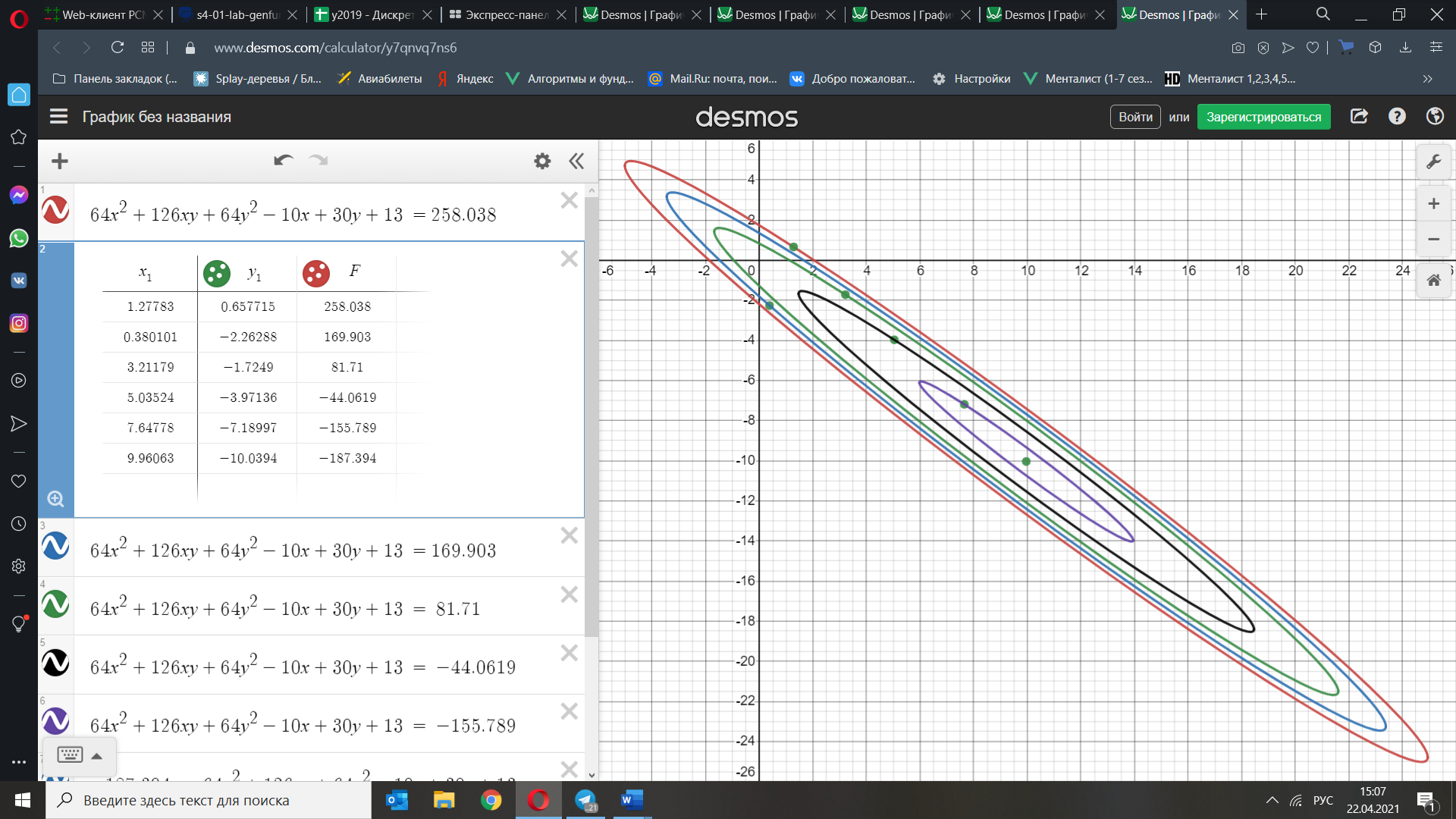
(25 итераций)

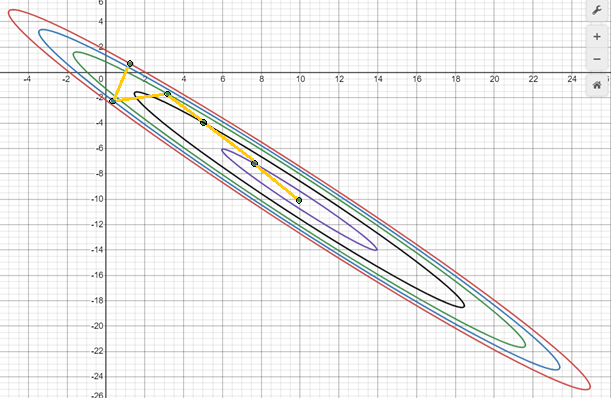




* метод градиентного спуска;

(945 итераций)



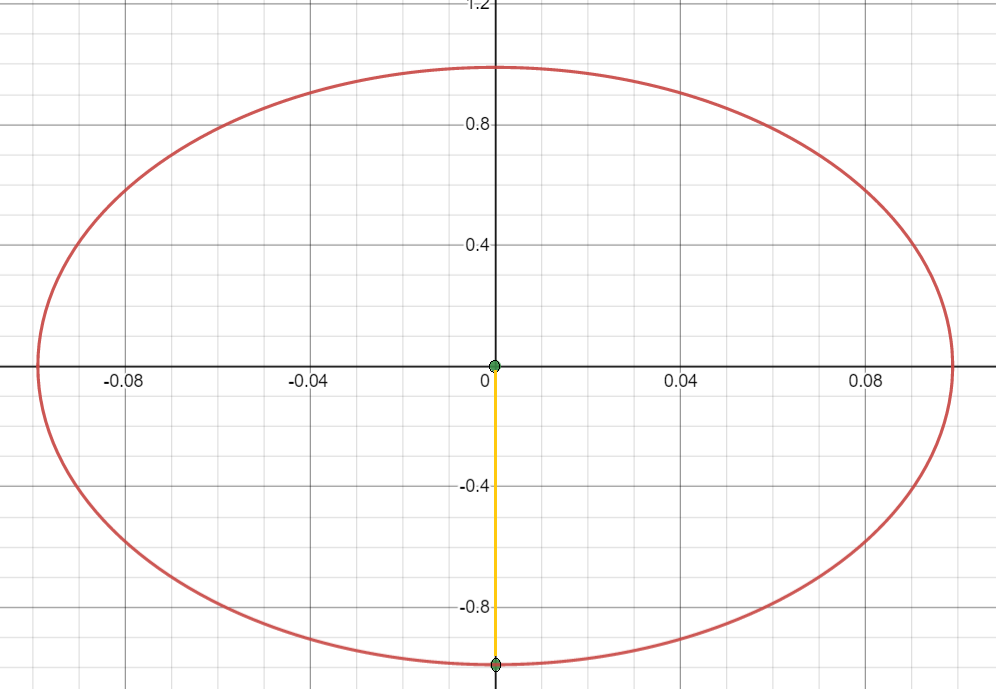




2. f(x, y) = 100x2+y2+300

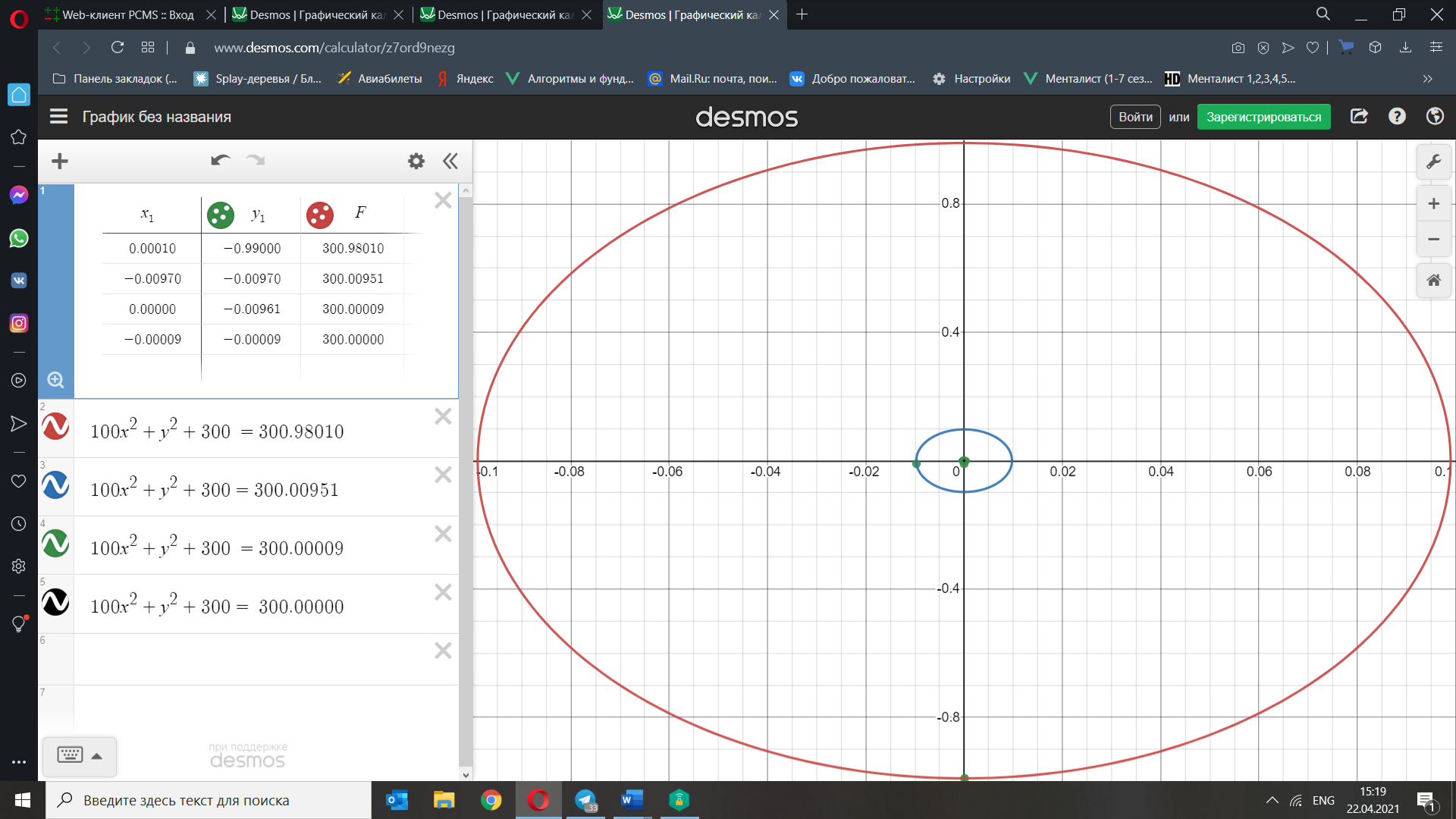
* метод сопряженных градиентов.

(2 итерации)



* метод наискорейшего спуска;

(4 итерации)

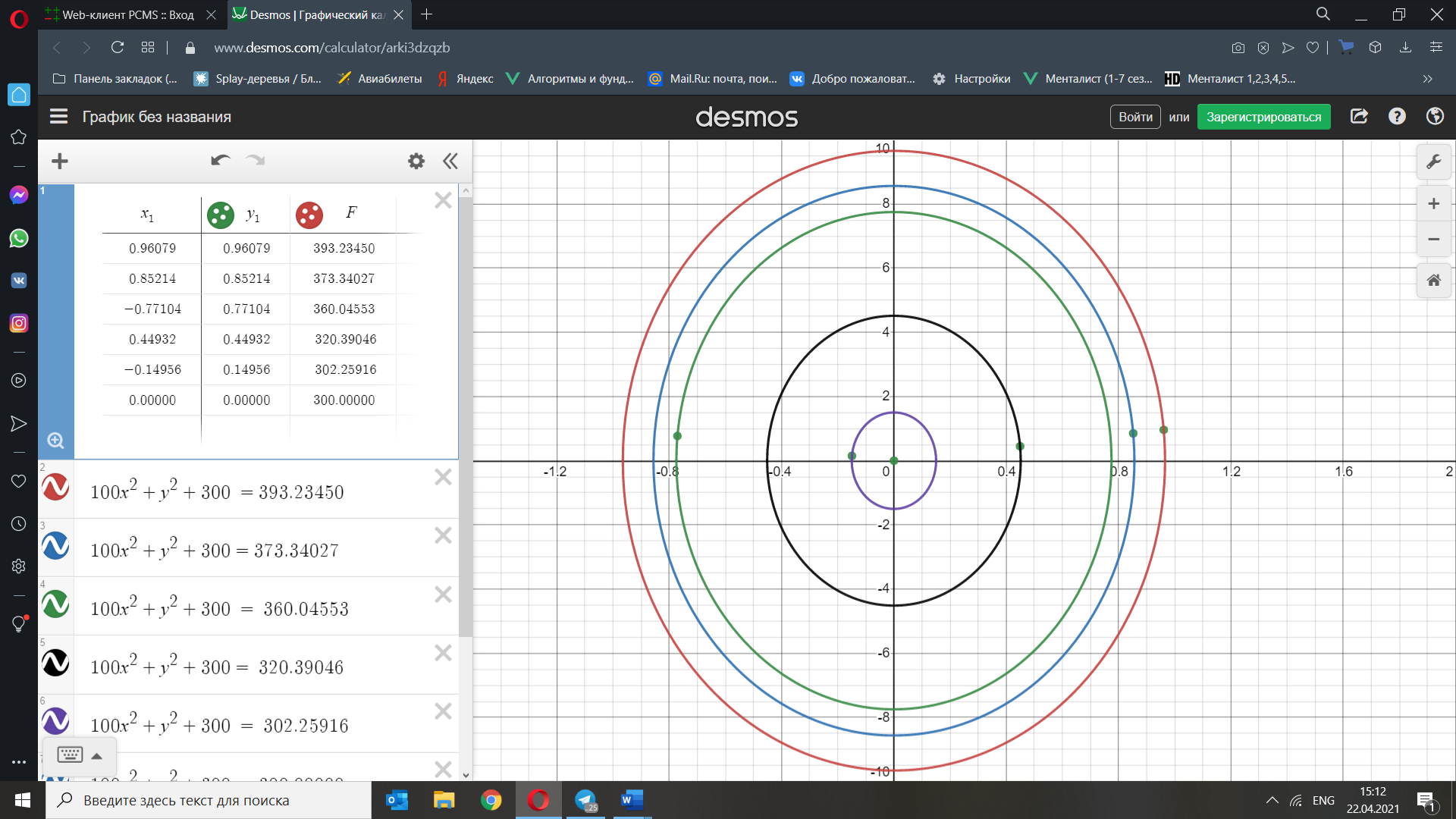


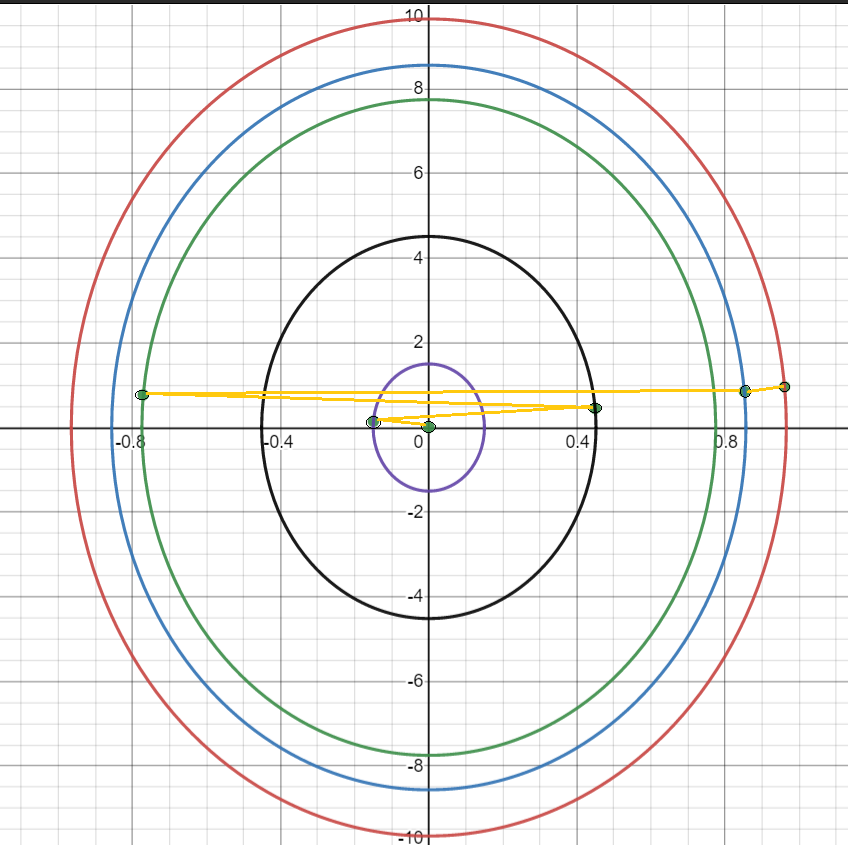




* метод градиентного спуска

(727 итераций)





3.

* метод сопряженных градиентов.

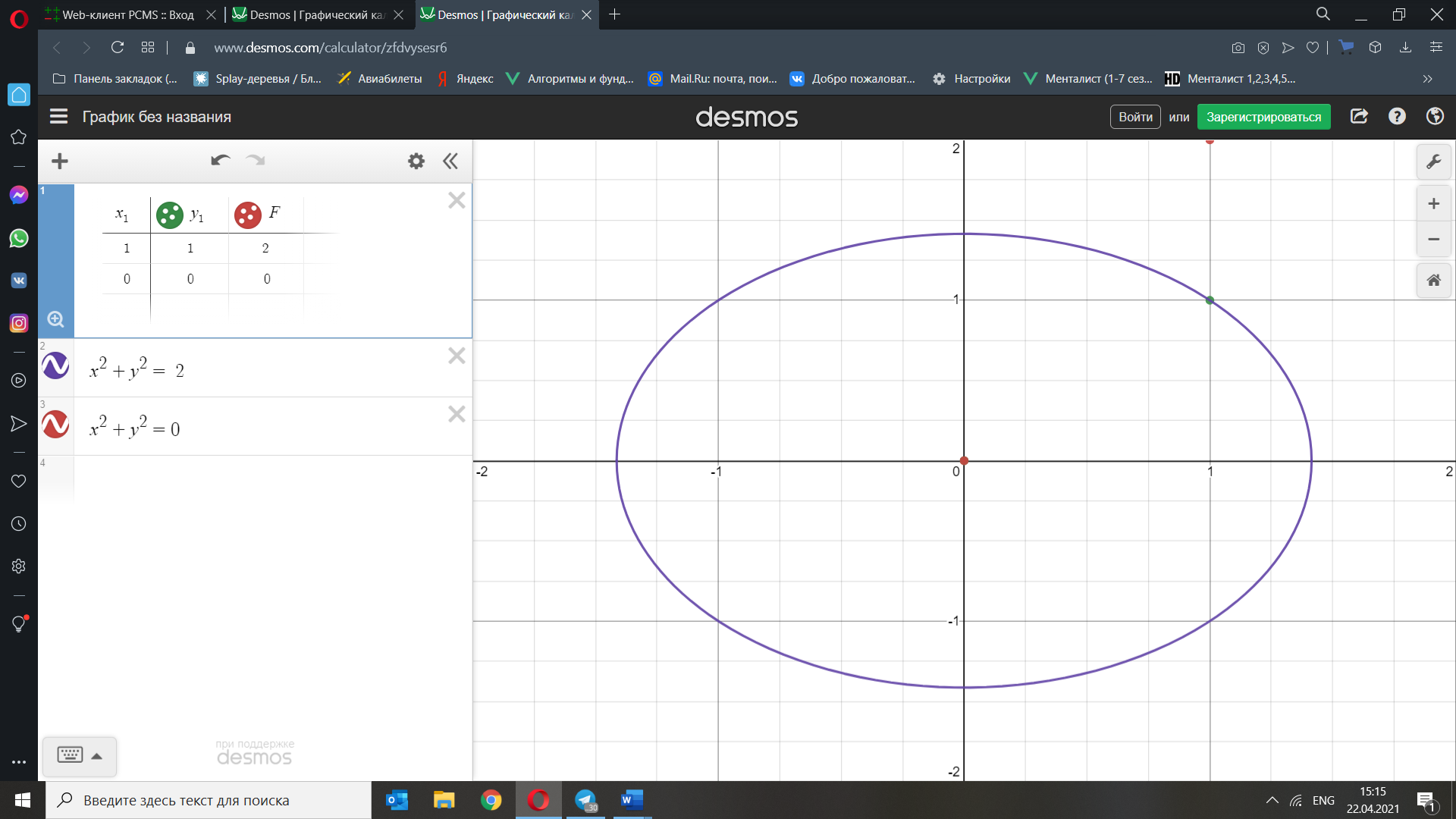
Найдено за одну итерацию

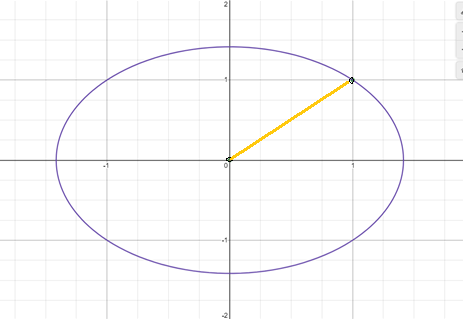
* метод наискорейшего спуска;

Найдено за одну итерацию

* метод градиентного спуска.

(2 итерации)





Выводы:

1. Количество итераций в методе наискорейшего спуска не зависит от выбора метода одномерной оптимизации.

2. Количество итераций метода сопряженных градиентов не превышает размерность пространства. Количество итераций градиентного и наискорейшего спуска не зависит от размерности пространства.

3. Количество итераций градиентного спуска и метода наискорейшего спуска увеличивается с числом обусловленности.

Приложение:

#include <iostream>

#include <vector>

#include <cmath>

#include <iomanip>

#include <fstream>;

#include "ConsoleApplication1.h"

using namespace std;

const double EPS1 = 0.00000001;

const double EPS2 = 0.00000001;

const double EPS3 = 0.00000001;

const double GOLDEN\_RATIO1 = (sqrt(5) - 1) / 2;

const double GOLDEN\_RATIO2 = (3 - sqrt(5)) / 2;

int N;

int M;

ofstream fout("sas.txt");

ofstream tout("FastestDescentBrentIterations.txt");

// ==================================================Дополнительные структуры и связааные с ними функции===============================================

struct Vector {

vector<double> values;

Vector(){ }

Vector(vector<double> v) : values(v) {}

Vector operator\*(const double& a) {

Vector vec(values);

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

vec.values[i] \*= a;

}

return vec;

}

const Vector operator+(const Vector& a) {

Vector vec(values);

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

vec.values[i] += a.values[i];

}

return vec;

}

const Vector operator-(const Vector& a) {

Vector vec(values);

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

vec.values[i] -= a.values[i];

}

return vec;

}

friend ostream& operator<< (ostream& os, Vector a) {

for (auto v : a.values) {

os << v << " ";

}

return os;

}

};

Vector coefficients;

double scalar\_mult(Vector x, Vector y) {

double res = 0;

for (int i = 0; i < x.values.size(); ++i) {

res += x.values[i] \* y.values[i];

}

return res;

}

double norm(Vector a) {

return sqrt(scalar\_mult(a, a));

}

struct Function {

double (\*function)(Vector x);

double fun\_calls = 0;

vector <vector<double> > hessian;

Vector b;

};

Vector hessian\_vector\_mult(vector<vector<double>> H, Vector A) {

vector <double> res;

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

double res\_value = 0;

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

res\_value += (H[i][j] \* A.values[j]);

}

res.push\_back(res\_value);

}

return Vector(res);

}

Vector gradient(Function fun, Vector x) {

return hessian\_vector\_mult(fun.hessian, x) + fun.b;

}

vector<vector<double> > make\_hessian(Vector x) {

vector <vector<double> > hessian(x.values.size(), vector<double>(x.values.size(), 0));

for (int i = 0; i < x.values.size(); ++i) {

hessian[i][i] = x.values[i] \* 2;

}

return hessian;

}

// =============================================Минимизируемые функции для задания 2=================================================================================

double function1(Vector x) {

return pow(x.values[0], 2) + pow(x.values[1], 2);

}

double function2(Vector x) {

return 100 \* pow(x.values[0], 2) + pow(x.values[1], 2) + 300;

}

double function3(Vector x) {

return 64 \* pow(x.values[0], 2) + 126 \* x.values[0] \* x.values[1] + 64 \* pow(x.values[1], 2) - 10 \* x.values[0] + 30 \* x.values[1] + 13;

}

//==============================================Минимизируемые функции для задания 3===================================================================

Vector generate\_offset\_vector() {

vector<double> result;

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

result.push\_back(rand() % 1000);

}

return Vector(result);

}

Vector generate\_coefficients\_vector(int m, int M) {

vector <double> result;

result.push\_back(m);

for (int i = 1; i < N - 1; ++i) {

result.push\_back(rand() % (M - m) + m);

}

result.push\_back(M);

return Vector(result);

}

double function\_for\_num(Vector x, double num) {

double res = 0;

for (int i = 0; i < x.values.size() - 1; ++i) {

res += coefficients.values[i] \* pow(x.values[i], 2);

}

res += num \* pow(x.values[x.values.size() - 1], 2);

return res;

}

//==============================================Методы одномерной оптимизации==========================================================================

double dychotomy(double a, double b, double(\*function)(double a, Vector x, Vector gradient, Function fun), Vector cur\_vec, Vector gradient, Function fun, int& fun\_calls) {

double delta = EPS2 / 2;

double x1, x2, func\_value1, func\_value2;

while ((b - a) / 2 > EPS2) {

x1 = (b + a - delta) / 2;

x2 = (b + a + delta) / 2;

func\_value1 = function(x1, cur\_vec, gradient, fun);

++fun\_calls;

func\_value2 = function(x2, cur\_vec, gradient, fun);

++fun\_calls;

if (func\_value1 <= func\_value2) {

b = x2;

}

else {

a = x1;

}

}

return (a + b)/2;

}

double golden\_ratio(double a, double b, double(\*function)(double a, Vector x, Vector gradient, Function fun), Vector cur\_vec, Vector gradient, Function fun, int& fun\_calls) {

double x1 = a + GOLDEN\_RATIO2 \* (b - a);

double x2 = a + GOLDEN\_RATIO1 \* (b - a);

double func\_value1 = function(x1, cur\_vec, gradient, fun);

double func\_value2 = function(x2, cur\_vec, gradient, fun);

fun\_calls += 2;

double const tau = GOLDEN\_RATIO1;

while ((b - a) / 2 > EPS2) {

if (func\_value1 <= func\_value2) {

b = x2;

x2 = x1;

func\_value2 = func\_value1;

x1 = b - tau \* (b - a);

func\_value1 = function(x1, cur\_vec, gradient, fun);

++fun\_calls;

}

else {

a = x1;

x1 = x2;

func\_value1 = func\_value2;

x2 = a + tau \* (b - a);

func\_value2 = function(x2, cur\_vec, gradient, fun);

++fun\_calls;

}

};

return (a + b) /2;

}

double sign(double x) {

if (x == 0) {

return 0;

}

if (x > 0)

return 1;

return -1;

}

double make\_parabola\_approximation(double x1, double x2, double x3, double func\_value1, double func\_value2, double func\_value3) {

double a0 = func\_value1;

double a1 = (func\_value2 - func\_value1) / (x2 - x1);

double a2 = (1 / (x3 - x2)) \* ((func\_value3 - func\_value1) / (x3 - x1) - (func\_value2 - func\_value1) / (x2 - x1));

return 0.5 \* (x1 + x2 - a1 / a2);

}

double Brent(double a, double c, double(\*function)(double a, Vector x, Vector gradient, Function fun), Vector cur\_vec, Vector gradient, Function fun, int& fun\_calls) {

const double K = GOLDEN\_RATIO2;

double x, w, v;

x = w = v = a + K \* (c - a);

double f\_x, f\_w, f\_v;

f\_x = f\_w = f\_v = function(x, cur\_vec, gradient, fun);

++fun\_calls;

double d = c - a, e = d;

double u;

while (true) {

double g = e;

e = d;

bool u\_accepted = false;

double tol = EPS2 \* abs(x) + EPS2 / 10;

if (abs(x - (a + c) / 2) + (c - a) / 2 <= 2 \* tol) {

break;

}

if (x != w && w != v && v != x && f\_x != f\_w && f\_w != f\_v && f\_v != f\_x) {

u = make\_parabola\_approximation(x, w, v, f\_x, f\_w, f\_v);

if (u >= a && u <= c && abs(u - x) < g / 2) {

//accept u

u\_accepted = true;

if (u - a < 2 \* tol || c - u < 2 \* tol) {

u = x - sign(x - (a + c) / 2) \* tol;

}

}

}

if (!u\_accepted) {//(true) { //parabol not accepted

if (x < (a + c) / 2) {

u = x + K \* (c - x);

e = c - x;

}

else {

u = x - K \* (x - a);

e = x - a;

}

}

if (abs(u - x) < tol) {

u = x + sign(u - x) \* tol;

}

d = abs(u - x);

double f\_u = function(u, cur\_vec, gradient, fun);

++fun\_calls;

if (f\_u <= f\_x) {

if (u >= x) {

a = x;

}

else {

c = x;

}

v = w;

w = x;

x = u;

f\_v = f\_w;

f\_w = f\_x;

f\_x = f\_u;

}

else {

if (u >= x) {

c = u;

}

else {

a = u;

}

if (f\_u <= f\_w || w == x) {

v = w;

f\_v = f\_w;

w = u;

f\_w = f\_u;

}

else if (f\_u <= f\_v || v == x || v == w) {

v = u;

f\_v = f\_u;

}

}

}

return (c + a) / 2;

}

//==============================================Метод градиентного спуска=====================================================================================

double gradient\_descent\_method(Vector x, Function fun, double alpha, int& iterations, int& fun\_call\_num) {

double f\_x = fun.function(x);

++fun\_call\_num;

Vector grad = gradient(fun, x);

Vector b = x;

double f\_b = f\_x;

while (norm(grad) >= EPS1) {

tout << "alpha = " << alpha << '\n';

++iterations;

grad = gradient(fun, x);

b = x - grad \* alpha;

f\_b = fun.function(b);

++fun\_call\_num;

if (norm(b - x) < EPS2 || abs(f\_b - f\_x) < EPS3) {

break;

}

if (f\_b < f\_x) {

x = b;

f\_x = f\_b;

fout << "x: {" << x << "}\n";

fout << "f\_x: {" << f\_x << "}\n";

fout << "====================================\n";

}

else {

alpha /= 2.0;

}

}

return f\_x;

}

//===============================================================Метод наискорейшего спуска============================================================

double fastest\_descent\_method(Vector a, Function fun, int& iterations, int& fun\_calls) {

double f\_a = fun.function(a);

Vector grad = gradient(fun, a);

while (norm(grad) >= EPS1)

{

++iterations;

double alpha = Brent(0, 100, [](double a, Vector x, Vector gradient, Function fun) {

return fun.function(x - gradient \* a);

}, a, grad, fun, fun\_calls);

Vector b = a - grad \* alpha;

fun\_calls += 2;

if (norm(b - a) < EPS2 || abs(fun.function(b) - fun.function(a)) < EPS3)

break;

a = b;

grad = gradient(fun, a);

fout << "x: {" << a << "}\n";

fout << "f\_x: {" << fun.function(a) << "}\n";

fout << "====================================\n";

}

return fun.function(a);

}

//==============================================================Метод сопряжённых градиентов=============================================================

double conjugated\_gradients\_method(Vector x, Function fun, int& iterations, int& fun\_calls) {

Vector grad = gradient(fun, x);

Vector p = grad \* (-1);

while (norm(grad) >= EPS1) {

++iterations;

Vector A = hessian\_vector\_mult(fun.hessian, p);

double alpha = pow(norm(grad), 2) / scalar\_mult(A, p);

fun\_calls += 2;

x = x + p \* alpha;

Vector new\_grad = grad + A \* alpha;

double beta = pow(norm(new\_grad), 2) / pow(norm(grad), 2);

grad = new\_grad;

p = new\_grad \* (-1) + p \* beta;

fout << "x: {" << x << "}\n";

fout << "f\_x: {" << fun.function(x) << "}\n";

fout << "====================================\n";

}

return fun.function(x);

}

//===================================================================

int main() {

Function fun1 = { function1, 0, make\_hessian(Vector(vector<double>({ 2, 2 }))), vector<double>(2, 0) };

Function fun2 = { function2, 0, make\_hessian(Vector(vector<double>({ 200, 2 }))), vector <double>(2, 0) };

Function fun3 = { function3, 0, vector<vector<double>>({ { 128,126 },{ 126, 128 } }), vector <double>({ -10 , 30 }) };

fout << fixed << setprecision(5);

cout << fixed << setprecision(5);

//cout << fixed << setprecision(5)<< gradient\_descent\_method(Vector(vector<double>({ 1, 1 })), fun2, 1.0/128) << '\n';

//cout << gradient\_descent\_method(Vector(vector<double>({ 1, 1 })), fun2, 1.0 / 101) << '\n';

//cout << fastest\_descent\_method(Vector(vector<double>({ -1, -1 })), fun3) << '\n';

//cout << conjugated\_gradients\_method(Vector(vector<double>({ 1, 1 })), fun3);

for (int n = 10; n <= 1000; n \*= 10) {

N = n;

tout << "Размерность векторов - " << n << '\n';

Vector vec(vector<double>(n, 1));

for (int i = 10; i < 1000; i += 100) {

coefficients = generate\_coefficients\_vector(1, i);

tout << "Число обусловленности - " << 2 \* i << '\n';

int iterations = 0;

int fun\_call\_num = 0;

Function fun = { [](Vector x) {

return function\_for\_num(x,N);

}, 0, make\_hessian(coefficients), vector<double>(n, 0) };

cout << fastest\_descent\_method(vec, fun, iterations, fun\_call\_num) << '\n';

cout << i << '\n';

cout << n << '\n';

tout << "Число вызовов функции = " << fun\_call\_num << '\n';

tout << "Число итераций = " << iterations << '\n';

}

tout << "\n";

}

}