|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | | |
|  | | | |
| Кафедра теоретической и прикладной информатики | | | |
|  | | | |
| Лабораторная работа № 2 | | | |
| по дисциплине «Методы оптимизации» | | | |
| **Методы спуска (0-го, 1-го и 2-го порядка и переменной метрики)** | | | |
|  | | | |
|  | 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 с методами поиска минимума функции n переменных в оптимизационных задачах без ограничений [1,5,7,8].

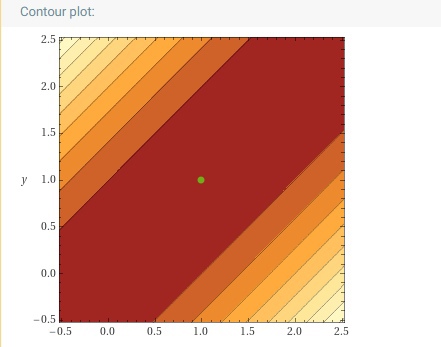
1. **Задание**
2. Реализовать два метода поиска экстремума функции:

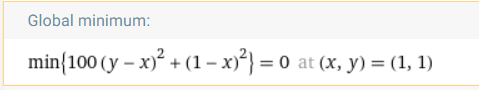
*Метод сопряженных градиентов в модификации Данилина-Пшеничного(Полака-Рибьера);*

*Метод Ньютона.*

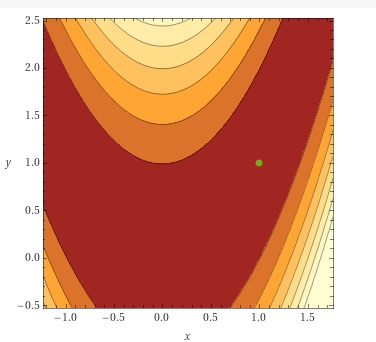
Включить в реализуемый алгоритм собственную процедуру, реализующую одномерный поиск по направлению.

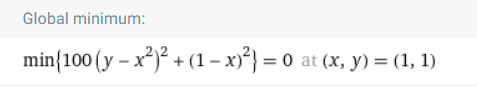
1. С использованием разработанного программного обеспечения исследовать алгоритмы на квадратичной функции





функции Розенброка





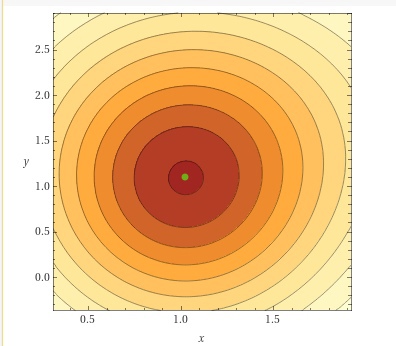
и на заданной в соответствии с вариантом тестовой функции, осуществляя спуск из различных исходных точек (не менее двух). Исследовать сходимость алгоритма, фиксируя точность определения минимума/максимума, количество итераций метода и количество вычислений функции в зависимости от задаваемой точности поиска.

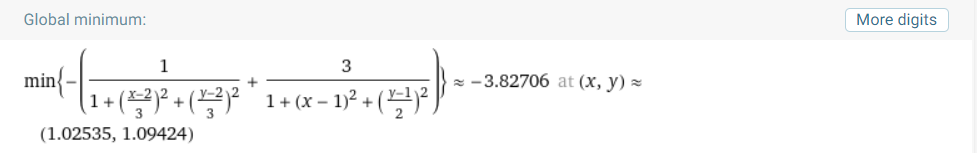
1. **Условие задачи**

Найти максимум заданной функции:

Для четных вариантов целевая функция имеет вид:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| № варианта |  |  |  |  |  |  |  |  |  |  |
| 2 | 1 | 3 | 2 | 1 | 3 | 1 | 2 | 1 | 3 | 2 |



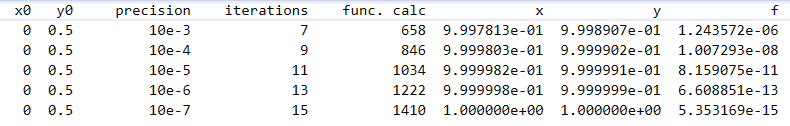


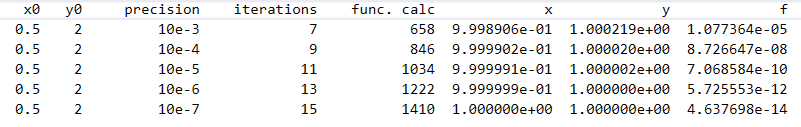
1. **Исследования реализованных алгоритмов на квадратичной функции, функции Розенброка и на тестовой функции**

*Метод Ньютона:*

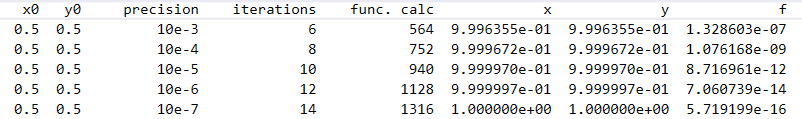
Квадратичная функция:

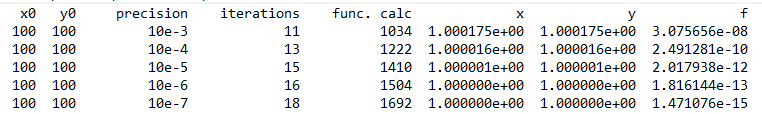
Точки с разных уровней:



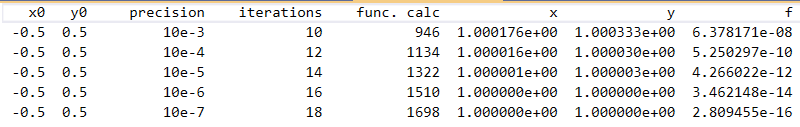


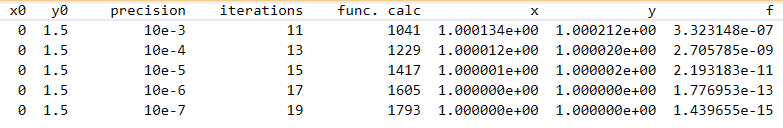
Точки на прямой симметрии:

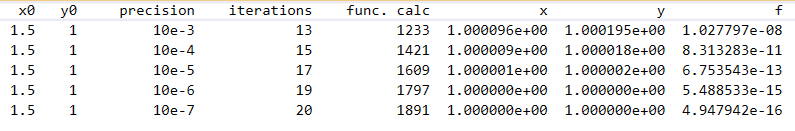




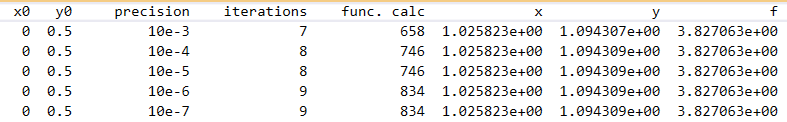
Функция Розенброка:

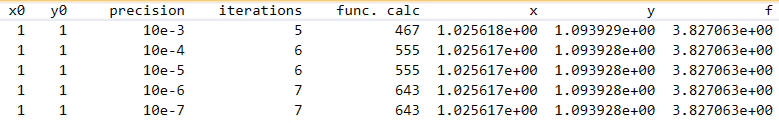


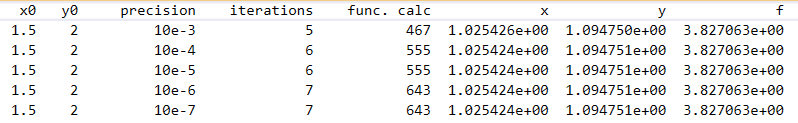




Тестовая функция:



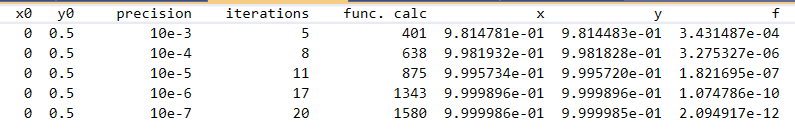


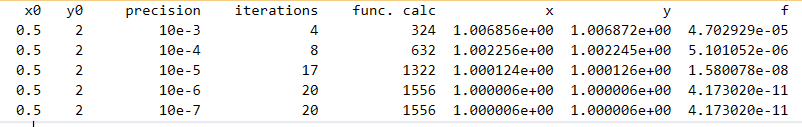


*Метод сопряженных градиентов в модификации Данилина-Пшеничного:*

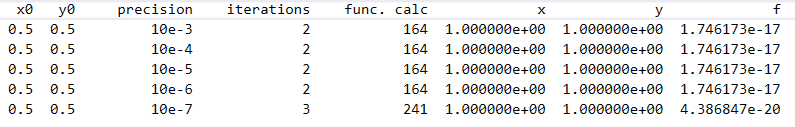
Квадратичная функция:

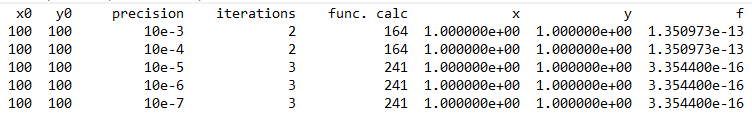
Точки с разных уровней:





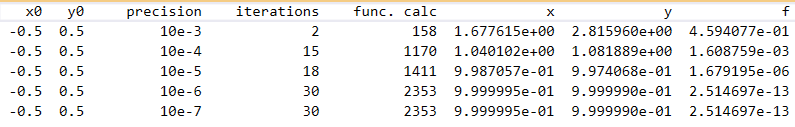
Точки на прямой симметрии:

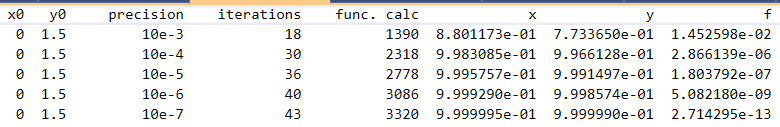


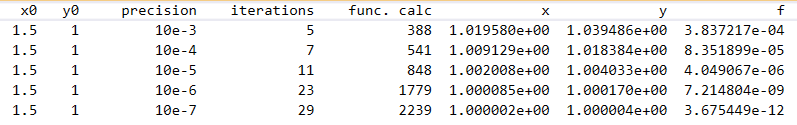


Можно заметить, что при нахождении точек начального приближения на линии симметрии уровней количество итераций метода практически не меняется, так как градиент в этих точках равен друг другу, совпадают и векторы направления поиска решения, меняется лишь коэффициент лямбда одномерного поиска, который, в свою очередь, находится достаточно точно.

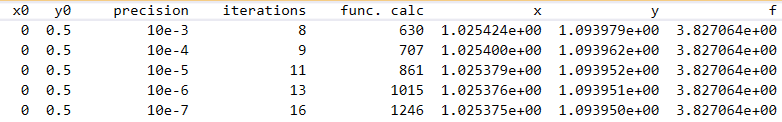
Функция Розенброка:

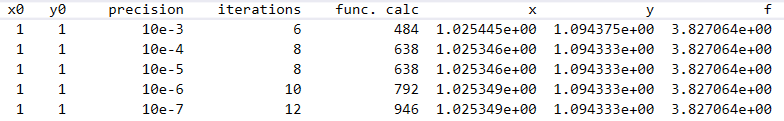


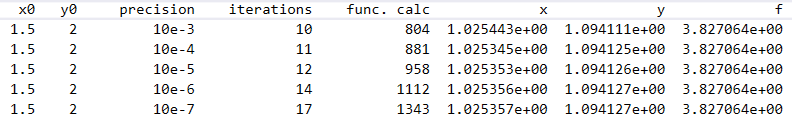




Тестовая функция:







Метод сопряженных градиентов должен сходиться за 2 итерации для квадратичных функций. Этого не происходит потому что вычисления градиентов функций происходят численно, точность нахождения градиента сильно зависит от параметра h в формулах расчета производных. Также строит отметить, что этот параметр варьируется от функции к функции, так, в квадратичной и тестовой функции h был взят порядка 10-1 в то время как для функции Розенброка потребовалось взять h=10-10.

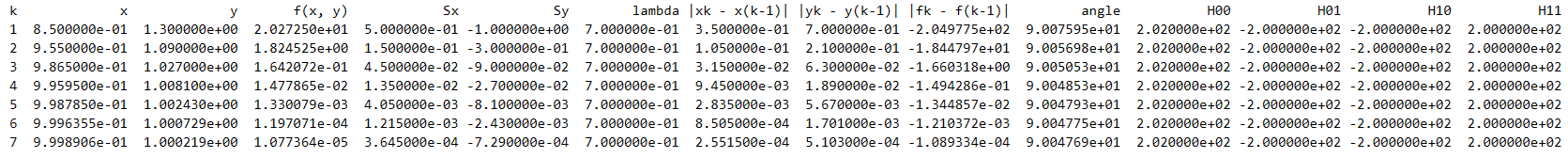
В методе Ньютона наблюдается аналогичная проблема, однако появляется дополнительная погрешность в связи с численным расчетом матрицы вторых производных.

1. **Исследования на сходимость реализованных алгоритмов**

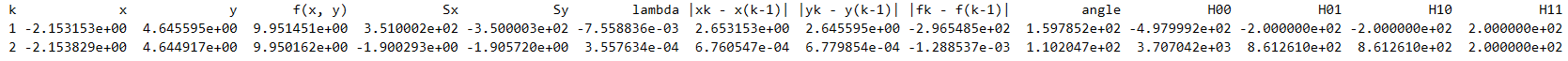
Начальная точка (0.5,2.0):

*Метод Ньютона:*

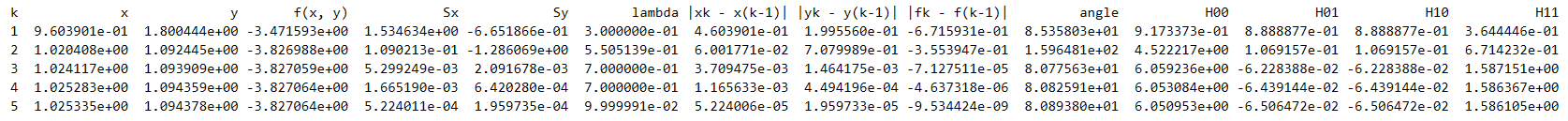
Квадратичная функция:



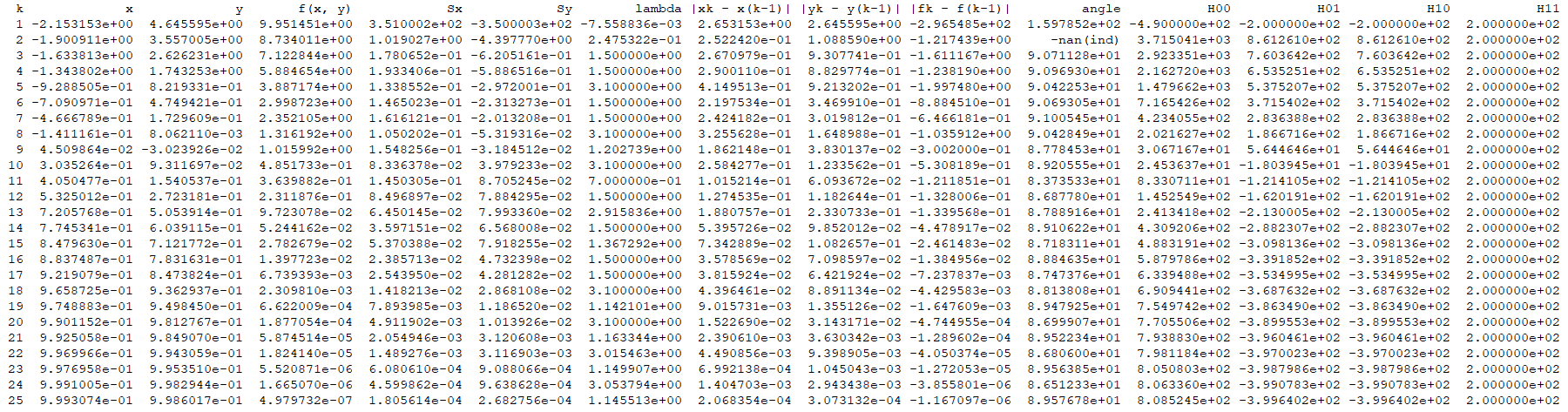
Функция Розенборка:



Тестовая функция:

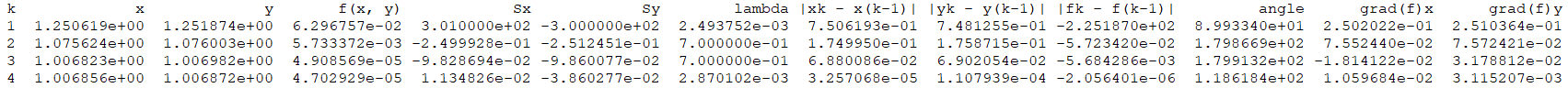


Заметим, что метод Ньютона не нашел минимум функции Розенброка в точке (1;1). Связано это с тем, что параметр h при расчете производных для матрицы Гёссе был взят порядка 1-3 для всех функций. Взяв h=1-1, можно получить следующий результат для функции Розенброка:



*Метод сопряженных градиентов в модификации Данилина-Пшеничного:*

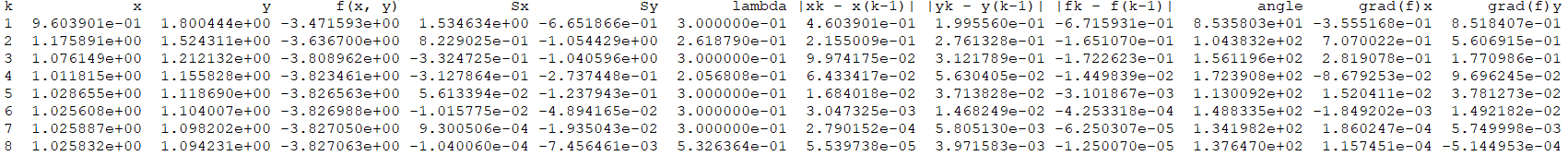
Квадратичная функция:



Функция Розенборка:



Тестовая функция:



**6 Выводы**

Метод Ньютона оказался быстрее метода сопряженных градиентов для функции Розенброка и тестовой функции, но оказался менее эффективным для первой функции. Стоит также отметить, что метод Ньютоне имеет большее число вычисления функции, как минимум из-за численного расчета матрицы Гёссе. В целом большое число вычислений функций объясняется вызовом метода одномерного поиска и функции поиска отрезка с минимумом по направлению на каждой итерации метода поиска экстремума.

**7 Программа**

*Файл «main.cpp»*

#include <iostream>

#include <vector>

#include <iomanip>

#include "DanPshen.h"

#include "Newton.h"

using namespace std;

int main()

{

vector<double> x0 = { 2, 5 };

ofstream \_f("f.txt");

ofstream fout;

DanPshen dp(2);

// Метод Данилина-Пшеничного

// На первой функции

fout = ofstream("results/f1\_pshen.txt");

fout << setw(14) << "x0" << setw(14) << "precision" << setw(14) << "iterations";

fout << setw(14) << "func. calc" << setw(14) << "x" << setw(14) << "y" << setw(14) << "f" << endl;

for(int i = -3; i >= -7 ; i--)

{

int iter\_count = dp.FindExtremum(f1, FindMinArgGolden, x0, pow(10, i), pow(10, i), pow(10, -1), \_f);

fout << setw(14) << "x0 : (2, 5)" << setw(12) << "10e" << i << setw(14) << iter\_count << setw(14) << dp.f\_calc\_count;

fout << setw(14) << dp.xk[0] << setw(14) << dp.xk[0] << setw(14) << f1(dp.xk) << endl;

}

fout.close();

// На второй функции

fout = ofstream("results/f2\_pshen.txt");

fout << setw(14) << "x0" << setw(14) << "precision" << setw(14) << "iterations";

fout << setw(14) << "func. calc" << setw(14) << "x" << setw(14) << "y" << setw(14) << "f" << endl;

for(int i = -3; i >= -7; i--)

{

int iter\_count = dp.FindExtremum(f2, FindMinArgGolden, x0, pow(10, i), pow(10, i), pow(10, -10), \_f);

fout << setw(14) << "x0 : (2, 5)" << setw(12) << "10e" << i << setw(14) << iter\_count << setw(14) << dp.f\_calc\_count;

fout << setw(14) << dp.xk[0] << setw(14) << dp.xk[0] << setw(14) << f1(dp.xk) << endl;

}

fout.close();

// На третьей функции

fout = ofstream("results/f3\_pshen.txt");

fout << setw(14) << "x0" << setw(14) << "precision" << setw(14) << "iterations";

fout << setw(14) << "func. calc" << setw(14) << "x" << setw(14) << "y" << setw(14) << "f" << endl;

for(int i = -3; i >= -7; i--)

{

int iter\_count = dp.FindExtremum(f3, FindMaxArgGolden, x0, pow(10, i), pow(10, i), pow(10, -1), \_f);

fout << setw(14) << "x0 : (2, 5)" << setw(12) << "10e" << i << setw(14) << iter\_count << setw(14) << dp.f\_calc\_count;

fout << setw(14) << dp.xk[0] << setw(14) << dp.xk[0] << setw(14) << f1(dp.xk) << endl;

}

fout.close();

Newton nt;

// Метод Ньютона

// На первой функции

fout = ofstream("results/f1\_newton.txt");

fout << setw(14) << "x0" << setw(14) << "precision" << setw(14) << "iterations";

fout << setw(14) << "func. calc" << setw(14) << "x" << setw(14) << "y" << setw(14) << "f" << endl;

for(int i = -3; i >= -7; i--)

{

int iter\_count = nt.FindExtremum(f1, FindMinArgGolden, x0, pow(10, i), pow(10, i), pow(10, -1), \_f);

fout << setw(14) << "x0 : (2, 5)" << setw(12) << "10e" << i << setw(14) << iter\_count << setw(14) << nt.f\_calc\_count;

fout << setw(14) << dp.xk[0] << setw(14) << dp.xk[0] << setw(14) << f1(dp.xk) << endl;

}

fout.close();

//// На второй функции

fout = ofstream("results/f2\_newton.txt");

fout << setw(14) << "x0" << setw(14) << "precision" << setw(14) << "iterations";

fout << setw(14) << "func. calc" << setw(14) << "x" << setw(14) << "y" << setw(14) << "f" << endl;

for(int i = -3; i >= -7; i--)

{

int iter\_count = nt.FindExtremum(f2, FindMinArgGolden, x0, pow(10, i), pow(10, i), pow(10, -10), \_f);

fout << setw(14) << "x0 : (2, 5)" << setw(12) << "10e" << i << setw(14) << iter\_count << setw(14) << nt.f\_calc\_count;

fout << setw(14) << dp.xk[0] << setw(14) << dp.xk[0] << setw(14) << f1(dp.xk) << endl;

}

fout.close();

//// На третьей функции

fout = ofstream("results/f3\_newton.txt");

fout << setw(14) << "x0" << setw(14) << "precision" << setw(14) << "iterations";

fout << setw(14) << "func. calc" << setw(14) << "x" << setw(14) << "y" << setw(14) << "f" << endl;

for(int i = -3; i >= -7; i--)

{

int iter\_count = nt.FindExtremum(f3, FindMaxArgGolden, x0, pow(10, i), pow(10, i), pow(10, -1), \_f);

fout << setw(14) << "x0 : (2, 5)" << setw(12) << "10e" << i << setw(14) << iter\_count << setw(14) << nt.f\_calc\_count;

fout << setw(14) << dp.xk[0] << setw(14) << dp.xk[0] << setw(14) << f1(dp.xk) << endl;

}

fout.close();

\_f.close();

// Метод Данилина-Пшеничного

// На первой функции

fout = ofstream("results/table.txt");

fout << "Method: Danilin-Pshenichniy" << endl;

fout << "f1" << endl;

dp.FindExtremum(f1, FindMinArgGolden, x0, 0.001, 0.001, pow(10, -1), fout);

fout << "f2" << endl;

dp.FindExtremum(f2, FindMinArgGolden, x0, 0.001, 0.001, pow(10, -10), fout);

fout << "f3" << endl;

dp.FindExtremum(f3, FindMaxArgGolden, x0, 0.001, 0.001, pow(10, -1), fout);

fout << "Method: Newton" << endl;

fout << "f1" << endl;

nt.FindExtremum(f1, FindMinArgGolden, x0, 0.001, 0.001, pow(10, -1), fout);

fout << "f2" << endl;

nt.FindExtremum(f2, FindMinArgGolden, x0, 0.001, 0.001, pow(10, -10), fout);

fout << "f3" << endl;

nt.FindExtremum(f3, FindMaxArgGolden, x0, 0.001, 0.001, pow(10, -1), fout);

fout.close();

int asd = 1111;

}

*Файл «Lab2Data.h»*

#pragma once

#include <vector>

#include "Vector.h"

using namespace std;

const double SQRT5 = sqrt(5);

const double PI = 3.14159265;

double f1(const vector<double>& x)

{

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

}

void gradf1(const vector<double>& x, vector<double>& res)

{

res = { 202 \* x[0] - 2 - 200 \* x[1], -200 \* x[0] + 200 \* x[1] };

}

double f2(const vector<double>& x)

{

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

}

double f3(const vector<double>& x)

{

double A1 = 1, A2 = 3, a1 = 2, a2 = 1;

double b1 = 3, b2 = 1;

double c1 = 2, c2 = 1;

double d1 = 3, d2 = 2;

double brack\_1 = (x[0] - a1) / b1;

double brack\_2 = (x[1] - c1) / d1;

double brack\_3 = (x[0] - a2) / b2;

double brack\_4 = (x[1] - c2) / d2;

double half\_1 = A1 / (1 + brack\_1 \* brack\_1 + brack\_2 \* brack\_2);

double half\_2 = A2 / (1 + brack\_3 \* brack\_3 + brack\_4 \* brack\_4);

return half\_1 + half\_2;

}

double f4(const vector<double>& x)

{

return (x[0] - 2) \* (x[0] - 2) + (x[1] - 4) \* (x[1] - 4) + 1;

}

double f(const vector<double>& x)

{

return 2 \* x[0] + pow(x[1], 3);

}

// 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(0.1, 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;

}

// Поиск отрезка с максимумом функции

int FindSegmentWithMax(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 максимума функции вдоль направления методом золотого сечения

int FindMaxArgGolden(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 = FindSegmentWithMax(0.1, 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 «Newton.h»*

#pragma once

#include <vector>

#include "Lab2Data.h"

#include "Vector.h"

class Newton

{

public:

int size = 2; // Размерность вектора

vector<vector<double>> H; // Матрица Гёссе

vector<vector<double>> IH; // Обратная матрица Гёссе

vector<double> t; // Вспомогательный вектор для расчета Гессиана

vector<double> grad; // Гардиент на текущем шаге

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

vector<double> xk1; // Новое приближение

vector<double> Sk; // Напрвление поиска

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

Newton() : size(2)

{

H.resize(2, vector<double>(2));

IH.resize(2, vector<double>(2));

t.resize(2);

grad.resize(2);

xk.resize(2);

xk1.resize(2);

Sk.resize(2);

t.resize(2);

}

// Расчитать Гессиан для двумерной функции (при eps = 1 все нормально)

void CalcHessian(double funct(const vector<double>&), const vector<double>& point, const double& eps)

{

H[0][0] = (funct({ point[0] + 2 \* eps, point[1] }) - 2 \* funct({ point[0], point[1] }) + funct({ point[0] - 2 \* eps, point[1] })) / (4 \* eps \* eps);

H[1][0] = (funct({ point[0] + eps, point[1] + eps }) -

funct({ point[0] + eps, point[1] - eps }) -

funct({ point[0] - eps, point[1] + eps }) +

funct({ point[0] - eps, point[1] - eps })) / (4 \* eps \* eps);

H[0][1] = H[1][0];

H[1][1] = (funct({ point[0], point[1] + 2 \* eps }) - 2 \* funct({ point[0], point[1] }) + funct({ point[0], point[1] - 2 \* eps })) / (4 \* eps \* eps);

}

// Расчитать обратную матрицу Гессе

void CalcInverseHessian()

{

double a = H[0][0];

double b = H[0][1];

double c = H[1][0];

double d = H[1][1];

double q = 1.0 / (a \* d - b \* c);

IH[0][0] = q \* d;

IH[0][1] = q \* -b;

IH[1][0] = q \* -c;

IH[1][1] = q \* a;

}

void CalcGrad(double funct(const vector<double>&), const vector<double>& point, vector<double>& res, const double& grad\_eps)

{

res[0] = (funct({ point[0] + grad\_eps, point[1] }) - funct({point[0] - grad\_eps, point[1] })) / (2 \* grad\_eps);

res[1] = (funct({ point[0], point[1] + grad\_eps }) - funct({point[0], point[1] - grad\_eps })) / (2 \* grad\_eps);

}

// Матрично-векторное умножение

void MatVecMult(const vector<vector<double>>& matrix, const vector<double>& vec, vector<double>& res)

{

res.resize(2);

res[0] = matrix[0][0] \* vec[0] + matrix[0][1] \* vec[1];

res[1] = matrix[1][0] \* vec[0] + matrix[1][1] \* vec[1];

}

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, const double& grad\_eps,

ofstream& fout)

{

f\_calc\_count = 0;

fout << setw(3) << "k";

fout << setw(14) << "x" << setw(14) << "y" << setw(14) << "f(x, y)";

fout << setw(14) << "Sx" << setw(14) << "Sy" << setw(14) << "lambda";

fout << setw(14) << "|xk - x(k-1)|" << setw(14) << "|yk - y(k-1)|" << setw(14) << "|fk - f(k-1)|";

fout << setw(14) << "angle";

fout << setw(14) << "H00" << setw(14) << "H01" << setw(14) << "H10" << setw(14) << "H11" << endl;

xk = x0;

bool exit\_flag;

int iter\_count = 0;

do

{

CalcHessian(funct, xk, 1e-3);

f\_calc\_count += 11;

CalcInverseHessian();

if(!(IH[0][0] > 0 && IH[0][0] \* IH[1][1] - IH[1][0] \* IH[0][1] > 0))

{

IH[0][0] = 1;

IH[0][1] = 0;

IH[1][0] = 0;

IH[1][1] = 1;

}

CalcGrad(funct, xk, grad, grad\_eps);

f\_calc\_count += 4;

grad \*= -1;

MatVecMult(IH, grad, Sk);

// Минимизация функции f по направлению Sk

double lambda;

f\_calc\_count += min\_max(funct, xk, Sk, 1e-15, lambda) + 1;

// Получение нового приближения

xk1 = xk + lambda \* Sk;

// Блок вывода

fout << fixed << setw(3) << iter\_count + 1;

fout << scientific;

fout << setw(14) << xk1[0] << setw(14) << xk1[1] << setw(14) << funct(xk1);

fout << setw(14) << Sk[0] << setw(14) << Sk[1] << setw(14) << lambda;

fout << setw(14) << abs(xk1[0] - xk[0]) << setw(14) << abs(xk1[1] - xk[1]) << setw(14) << funct(xk1) - funct(xk);

fout << setw(14) << acos((xk1[0] \* Sk[0] + xk1[1] \* Sk[1]) / (Norm(xk1) \* Norm(grad))) \* 180 / PI;

fout << setw(14) << H[0][0] << setw(14) << H[0][1] << setw(14) << H[1][0] << setw(14) << H[1][1] << endl;

// 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(iter\_count < 1000 && exit\_flag == false);

fout << endl;

return iter\_count;

}

};

*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, const double& grad\_eps,

ofstream& fout)

{

f\_calc\_count = 0;

// 1. Расчет градиента функции f в точке x0

CalcGrad(funct, x0, Sk, grad\_eps);

f\_calc\_count += 4;

Sk \*= -1;

xk = x0;

fout << setw(3) << "k";

fout << setw(14) << "x" << setw(14) << "y" << setw(14) << "f(x, y)";

fout << setw(14) << "Sx" << setw(14) << "Sy" << setw(14) << "lambda";

fout << setw(14) << "|xk - x(k-1)|" << setw(14) << "|yk - y(k-1)|" << setw(14) << "|fk - f(k-1)|";

fout << setw(14) << "angle" << endl;

bool exit\_flag;

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-15, lambda) + 1;

// Получение нового приближения

xk1 = xk + lambda \* Sk;

// Блок вывода

fout << fixed << setw(3) << iter\_count + 1;

fout << scientific;

fout << setw(14) << xk1[0] << setw(14) << xk1[1] << setw(14) << funct(xk1);

fout << setw(14) << Sk[0] << setw(14) << Sk[1] << setw(14) << lambda;

fout << setw(14) << abs(xk1[0] - xk[0]) << setw(14) << abs(xk1[1] - xk[1]) << setw(14) << funct(xk1) - funct(xk);

fout << setw(14) << acos((xk1[0] \* Sk[0] + xk1[1] \* Sk[1]) / (Norm(xk1) \* Norm(Sk))) \* 180 / PI << endl;

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

CalcGrad(funct, xk1, grad\_xk1, grad\_eps);

f\_calc\_count += 4;

//gradf1(xk1, grad\_xk1);

//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);

fout << endl;

return iter\_count;

}

void CalcGrad(double funct(const vector<double>&), const vector<double>& point, vector<double>& res, const double& grad\_eps)

{

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

{

t = point;

t[i] += grad\_eps;

res[i] = funct(t);

t[i] = point[i] - grad\_eps;

res[i] -= funct(t);

res[i] /= 2.0 \* grad\_eps;

}

}

};