ФЕДЕРАЛЬНОЕ АГЕНТСВО ПО ОБРАЗОВАНИЮ РОССИЙСКОЙ ФЕДЕРАЦИИ

УФИМСКИЙ ГОСУДАРСТВЕННЫЙ АВИАЦИОННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Общенаучный факультет

Кафедра ВВТиС

Отчет по лабораторным работам №2

**По дисциплине «Методы оптимизации»**

Группа ПМИ-440

Студент \_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ Сайфутдинов Р.Ф.

(дата) (подпись) (Фамилия И.О.)

Проверил \_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_ Касаткин А. А.

(дата) (подпись) (Фамилия И.О.)

Уфа 2016

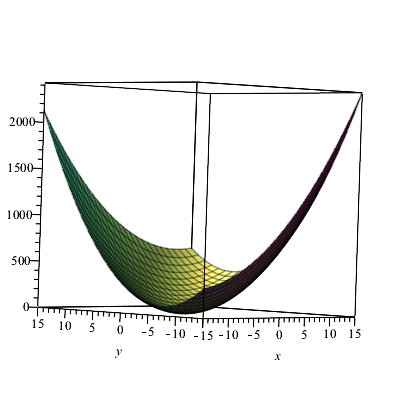
**Цель работы:** изучить численные методы поиска безусловного экстремума действительной функции *n* действительных переменных.

**Задание:**

Найти решение задачи безусловной оптимизации для заданной целевой функции, используя теоремы о необходимых и достаточных условиях экстремума.

Целевая функция: 

График целевой функции:



**Утверждение 1.** (необходимые условия экстремума второго порядка)

Пусть точка есть точка локального минимума (максимума) функциина множестве и дифференцируема в точке . Тогда градиент функции в точке равен нулю, т.е.

|  |  |
| --- | --- |
|  | (1) |

*или*

|  |  |
| --- | --- |
|  | (2) |

**Определение 1.** Точки , удовлетворяющие условию (1)или(2),называются стационарными.

**Утверждение 2.**(необходимые условия экстремума второго порядка)

Пусть точка есть точка локального минимума (максимума) функции на множестве и функция дважды дифференцируема в этой точке. Тогда матрица Гессе функции , вычисленная в точке , является положительно полуопределенной (отрицательно полуопределенной), т.е.

**Утверждение 3**. (достаточные условия экстремума).

Пусть функция , в точке дважды дифференцируема, её градиент равен нулю, а матрица Гессе является положительно полуопределенной (отрицательно полуопределенной) , т.е.

Тогда точка есть точка локального минимума(максимума) функции на множестве

Начало

Ввод:x0

k=0

dk=-[H(xk)+λkE]-1grad(f(xk))

xk+1=xk+dk

f(xk+1)<f(xk)

λk=2λk

λk+1=λk/2

k=k+1

Вывод:xk

Конец

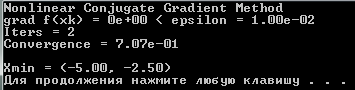
Блок-схема метода Марквардта

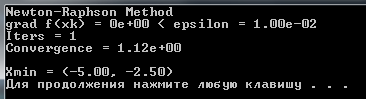
**Решение задачи в пакете Maple**

**> **



**Пример выполнения программы**





**Вывод:** метод сопряженных градиентов сходится быстрее, чем метод Ньютона-Рафсона.

**Листинг**

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <inttypes.h>

#include <stdbool.h>

#include <math.h>

typedef struct point2d {

double x, y;

} point2d\_t;

typedef struct mat2x2 {

double a11, a12, a21, a22;

} mat2x2\_t;

double dotProduct2d(point2d\_t x, point2d\_t y);

point2d\_t mat\_vec2d(double alpha, mat2x2\_t mat, point2d\_t x);

point2d\_t scalarmul\_vec2d(double alpha, point2d\_t x);

point2d\_t add\_vec2d(point2d\_t x, point2d\_t y);

double det\_mat2x2(mat2x2\_t mat);

mat2x2\_t inversed\_mat2x2(mat2x2\_t mat);

bool isPositiveDefMat2x2(mat2x2\_t mat);

double fun(point2d\_t x);

point2d\_t grad\_fun(point2d\_t x);

mat2x2\_t fun\_Hessian\_mat2x2();

double findT\_phi(point2d\_t x, point2d\_t grad\_fun\_x);

point2d\_t nonlinearConjugateGradientMethod(point2d\_t x0, double epsilon, uint32\_t maxIter);

point2d\_t methodNewtonRaphson(point2d\_t x0, double epsilon, uint32\_t maxIter);

int32\_t main() {

const point2d\_t x0 = { -4.0, -2.0 };

const double epsilon = 1.0e-2;

const uint32\_t maxIter = 100;

const uint8\_t test = 2; // 1 - Nonlinear Conjugate Gradient Method

// 2 - method Newton-Raphson

point2d\_t xmin;

switch (test) {

case 1:

printf("Nonlinear Conjugate Gradient Method\n");

xmin = nonlinearConjugateGradientMethod(x0, epsilon, maxIter);

break;

case 2:

printf("Newton-Raphson Method\n");

xmin = methodNewtonRaphson(x0, epsilon, maxIter);

break;

default:

printf("Error! value test\n");

system("pause");

return -1;

}

printf("\nXmin = (%.2f, %.2f)\n", xmin.x, xmin.y);

system("pause");

return 0;

}

inline double dotProduct2d(const point2d\_t x, const point2d\_t y) {

return x.x \* y.x + x.y \* y.y;

}

inline point2d\_t mat\_vec2d(const double alpha, const mat2x2\_t mat, const point2d\_t x) {

const point2d\_t vec = { alpha \* (mat.a11 \* x.x + mat.a12 \* x.y),

alpha \* (mat.a21 \* x.x + mat.a22 \* x.y) };

return vec;

}

inline point2d\_t scalarmul\_vec2d(const double alpha, const point2d\_t x) {

const point2d\_t vec = { alpha \* x.x, alpha \* x.y };

return vec;

}

inline point2d\_t add\_vec2d(const point2d\_t x, const point2d\_t y) {

const point2d\_t vec = { x.x + y.x, x.y + y.y };

return vec;

}

inline double det\_mat2x2(const mat2x2\_t mat) {

return mat.a11 \* mat.a22 - mat.a21 \* mat.a12;

}

inline mat2x2\_t inversed\_mat2x2(const mat2x2\_t mat) {

const double det\_mat = det\_mat2x2(mat);

if (det\_mat == 0.0) {

printf("Determinant = 0!\n");

const mat2x2\_t inv\_mat2x2 = { 0, 0, 0, 0 };

return inv\_mat2x2;

}

const double inv\_det\_mat = 1.0 / det\_mat;

const mat2x2\_t inv\_mat2x2 = { mat.a22 \* inv\_det\_mat, -mat.a12 \* inv\_det\_mat,

-mat.a21 \* inv\_det\_mat, mat.a11 \* inv\_det\_mat };

return inv\_mat2x2;

}

inline bool isPositiveDefMat2x2(const mat2x2\_t mat) {

// Sylvester's criterion

if (mat.a11 <= 0.0 || det\_mat2x2(mat) <= 0.0) {

printf("No Positive-definite matrix!\n");

return false;

} else {

return true;

}

}

inline double fun(const point2d\_t x) {

return (x.x - 2.0 \* x.y) \* (x.x - 2.0 \* x.y) + (x.x + 5.0) \* (x.x + 5.0);

}

inline point2d\_t grad\_fun(const point2d\_t x) {

const point2d\_t grad = { 4.0 \* x.x - 4.0 \* x.y + 10.0,

8.0 \* x.y - 4.0 \* x.x };

return grad;

}

inline mat2x2\_t fun\_Hessian\_mat2x2() {

const mat2x2\_t H = { 4.0, -4.0, -4.0, 8.0 };

return H;

}

inline double findT\_phi(const point2d\_t x, const point2d\_t d) {

// Quadratic function

//if (dotProduct2d(d, d) < 0) {

// printf("Error! d^phi/dt^2 < 0\n");

// return 0;

//}

const double A = (d.x - 2.0 \* d.y);

return -(A \* (x.x - 2.0 \* x.y) + (5.0 + x.x) \* d.x) / (A \* A + d.x \* d.x);

}

point2d\_t nonlinearConjugateGradientMethod(const point2d\_t x0, const double epsilon, const uint32\_t maxIter) {

const double epsilon2 = epsilon \* epsilon;

point2d\_t xk1 = x0;

point2d\_t xk, d;

double dot\_grad\_xk, convergence;

bool is\_seq = false;

uint32\_t k = 0;

for (k; k < maxIter; ++k) {

const point2d\_t grad\_fun\_xk1 = grad\_fun(xk1);

const double dot\_grad\_xk1 = dotProduct2d(grad\_fun\_xk1, grad\_fun\_xk1);

if (dot\_grad\_xk1 < epsilon2) {

printf("grad f(xk) = %.e < epsilon = %.2e\n", sqrt(dot\_grad\_xk1), epsilon);

break;

}

if (k == 0) {

d = scalarmul\_vec2d(-1.0, grad\_fun\_xk1);

} else {

const double beta = dot\_grad\_xk1 / dot\_grad\_xk;

d = add\_vec2d(scalarmul\_vec2d(beta, d), scalarmul\_vec2d(-1.0, grad\_fun\_xk1));

}

xk = xk1;

const double t = findT\_phi(xk, d);

const point2d\_t xk1\_minus\_xk = scalarmul\_vec2d(t, d);

xk1 = add\_vec2d(xk1, xk1\_minus\_xk);

convergence = dotProduct2d(xk1\_minus\_xk, xk1\_minus\_xk);

const double abs\_fk1\_minus\_fk = fabs(fun(xk1) - fun(xk));

if (convergence < epsilon2 && abs\_fk1\_minus\_fk < epsilon) {

if (is\_seq) {

break;

} else {

is\_seq = true;

}

} else {

is\_seq = false;

}

dot\_grad\_xk = dot\_grad\_xk1;

}

printf("Iters = %" PRIu32 "\nConvergence = %.2e\n", k, sqrt(convergence));

return xk1;

}

point2d\_t methodNewtonRaphson(const point2d\_t x0, const double epsilon, const uint32\_t maxIter) {

const double epsilon2 = epsilon \* epsilon;

point2d\_t xk1 = x0;

bool is\_seq = false;

uint32\_t k = 0;

double convergence;

for (k; k < maxIter; ++k) {

const point2d\_t grad\_fun\_xk1 = grad\_fun(xk1);

const double dot\_grad\_xk1 = dotProduct2d(grad\_fun\_xk1, grad\_fun\_xk1);

if (dot\_grad\_xk1 < epsilon2) {

printf("grad f(xk) = %.e < epsilon = %.2e\n", sqrt(dot\_grad\_xk1), epsilon);

break;

}

const mat2x2\_t H = fun\_Hessian\_mat2x2();

const mat2x2\_t invH = inversed\_mat2x2(H);

const point2d\_t d = isPositiveDefMat2x2(invH) ? mat\_vec2d(-1.0, invH, grad\_fun\_xk1)

: scalarmul\_vec2d(-1.0, grad\_fun\_xk1);

const point2d\_t xk = xk1;

const double t = findT\_phi(xk, d);

const point2d\_t xk1\_minus\_xk = scalarmul\_vec2d(t, d);

xk1 = add\_vec2d(xk1, xk1\_minus\_xk);

convergence = dotProduct2d(xk1\_minus\_xk, xk1\_minus\_xk);

const double abs\_fk1\_minus\_fk = fabs(fun(xk1) - fun(xk));

if (convergence < epsilon2 && abs\_fk1\_minus\_fk < epsilon) {

if (is\_seq) {

break;

} else {

is\_seq = true;

}

} else {

is\_seq = false;

}

}

printf("Iters = %" PRIu32 "\nConvergence = %.2e\n", k, sqrt(convergence));

return xk1;

}