ФЕДЕРАЛЬНОЕ АГЕНТСВО ПО ОБРАЗОВАНИЮ РОССИЙСКОЙ ФЕДЕРАЦИИ

УФИМСКИЙ ГОСУДАРСТВЕННЫЙ АВИАЦИОННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Общенаучный факультет

Кафедра ВВТиС

Отчет по лабораторным работам №3

**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 методы поиска условного экстремума действительной функции.

**Теоретическая часть**

Рассмотрим задачу поиска минимума функции

(3.1)

Множество допустимых решений задается следующими условиями

(3.2)

Функция называется *функцией Лагранжа*.

Пусть функции дважды непрерывно дифференцируемы. Необходимые и достаточные условия первого порядка решения приведенной задачи оптимизации с ограничениями в виде равенств и неравенств приведены в следующей теореме.

*Теорема 3.1.* Точка является локальным минимумом задачи условной оптимизации (3.1), (3.2) тогда и только тогда, когда

1. функции выпуклые;
2. функции линейные;
3. существуют такие числа , что выполняются следующие условия:

* условие стационарности функции Лагранжа по *x*
* условие допустимости решения
* условие неотрицательности
* условие дополняющей нежесткости

Множество называется множеством активных в точке ограничений-неравенств.

Пусть линейно независимы.

*Теорема 3.2.* Точка является точкой строгого локального минимума задачи условной оптимизации (3.1), (3.2) тогда и только тогда, когда

1. существует вектор такой, что вектор удовлетворяет условиям 3 теоремы 3.1;
2. для любого ненулевого вектора , удовлетворяющего системе:

выполняется неравенство , где – матрица

Для решения большинства практических задач условной оптимизации используются численные методы. Достаточно часто используются методы штрафных функций, общий принцип которых заключается в замене исходной задачи на решение последовательности задач оптимизации без ограничений. Строится вспомогательная функция такая, что приближенное решение задачи (3.1), (3.2) находится в результате решения последовательности задач безусловной минимизации данной функции. Рассмотрим алгоритмы нескольких методов штрафных функций.

*Метод внешних штрафов*

1. Задаем следующие значения:

*n* – размерность вектора *x*;

*m* – число ограничений-равенств;

*l* – число всех ограничений;

– точность решения задачи;

– точность решения задачи безусловной минимизации;

– номер итерации;

– начальное приближение для , задается вне множества допустимых решений ;

– начальное значение параметра штрафа, обычно выбирают ;

– число для увеличения параметра штрафа, обычно выбирают .

1. Задаем вспомогательную функцию , где штрафная функция

– срезка функции:

1. Найдем значение , доставляющее минимум функции по *x* при фиксированном с помощью одного из методов безусловной минимизации (лабораторные работы 1, 2). В качестве начальной точки использовать , в качестве параметра окончания – константу .
2. Вычислить Если то решение задачи найдено , в противном случае переходим к шагу 5.
3. Изменяем значения . Возвращаемся к шагу 2.

*Метод внутренних штрафов (барьерных функций)*

Данный метод используется в задачах с ограничениями-неравенствами, .

1. Задаем следующие значения:

*n* – размерность вектора *x*;

*l* – число всех ограничений;

– точность решения задачи;

– точность решения задачи безусловной минимизации;

– номер итерации;

– начальное приближение для , задается внутри множества допустимых решений ;

– начальное значение параметра штрафа, обычно выбирают ;

– число для уменьшения параметра штрафа, обычно выбирают .

1. Задаем вспомогательную функцию , где штрафная функция или
2. Найдем значение , доставляющее минимум функции по *x* при фиксированном с помощью одного из методов безусловной минимизации (лабораторные работы 1, 2). В качестве начальной точки использовать , в качестве параметра окончания – константу . Проверить принадлежность .
3. Вычислить Если то решение задачи найдено , в противном случае переходим к шагу 5.
4. Изменяем значения . Возвращаемся к шагу 2.

*Комбинированный метод штрафных функций*

В данном методе для ограничений типа равенств используется метод внешних штрафов, а для ограничений типа неравенств – метод внутренних штрафов.

1. Задаем следующие значения:

*n* – размерность вектора *x*;

*l* – число всех ограничений;

– точность решения задачи;

– точность решения задачи безусловной минимизации;

– номер итерации;

– начальное приближение для , задается так, чтобы строго выполнялись ограничения типа неравенств ;

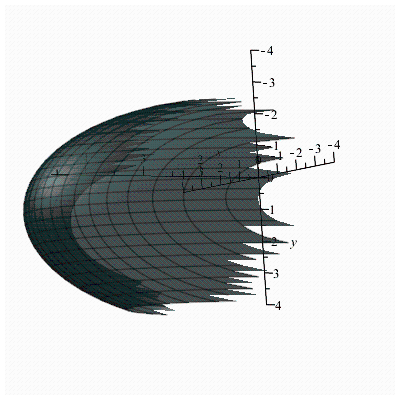
– начальное значение параметра штрафа, обычно выбирают ;

– число для уменьшения параметра штрафа.

1. Задаем вспомогательную функцию , где штрафная функция или . Можно использовать разные значения для внутренних и для внешних штрафов.
2. Найдем значение , доставляющее минимум функции по *x* при фиксированном с помощью одного из методов безусловной минимизации (лабораторные работы 1, 2). В качестве начальной точки использовать , в качестве параметра окончания – константу .
3. Вычислить Если то решение задачи найдено , в противном случае переходим к шагу 5.
4. Изменяем значения . Возвращаемся к шагу 2.

**Решение задачи в пакете Maple**

**> **





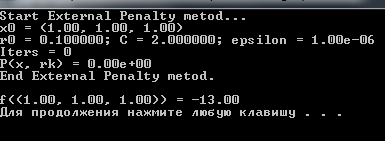
*Блок-схема метода:*

C, r, x, k,

НЕТ

ДА

**Пример выполнения программы**

****

**Вывод:** в ходе лабораторной работы были изучены методы поиска условного экстремума действительной функции и реализован метод внешних штрафов.**Листинг**

**main.h**

#ifndef MAIN\_H

#define MAIN\_H

#include <stdio.h>

#include <stdint.h>

#include <inttypes.h>

#include <stdbool.h>

#include <math.h>

#include <stdlib.h>

typedef struct segment {

double a;

double b;

} segment\_t;

typedef struct point3d {

double x[3];

} point3d\_t;

typedef struct mat3d {

double a[9];

} mat3d\_t;

inline double f(const point3d\_t x) {

double tmp = -13.0;

for (uint8\_t i = 0; i < 3; ++i) {

const double tmp2 = x.x[i] - 1.0;

tmp += tmp2 \* tmp2;

}

return tmp;

}

inline double g1(const point3d\_t x) {

return x.x[0] \* x.x[0] + x.x[1] \* x.x[1] + x.x[2] \* x.x[2] - 4.0;

}

inline double g1\_cut(const point3d\_t x) {

const double tmp = g1(x);

return tmp <= 0.0 ? 0.0 : tmp;

}

inline double g2(const point3d\_t x) {

return -x.x[2];

}

inline double g2\_cut(const point3d\_t x) {

const double tmp = g2(x);

return tmp <= 0.0 ? 0.0 : tmp;

}

inline double p\_ext(const point3d\_t x, const double rk) {

const double g1\_x = g1\_cut(x);

const double g2\_x = g2\_cut(x);

return rk \* 0.5 \* (g1\_x \* g1\_x + g2\_x \* g2\_x);

}

inline double p\_int(const point3d\_t x, const double rk) {

const double g1\_x = g1(x);

const double g2\_x = g2(x);

if (g1\_x >= 0 || g2\_x >= 0) {

return -(double)INFINITY;

}

return -rk \* (log(-g1\_x) + log(-g2\_x));

}

inline double dotProduct3d(const point3d\_t x, const point3d\_t y) {

return x.x[0] \* y.x[0] + x.x[1] \* y.x[1] + x.x[2] \* y.x[2];

}

inline point3d\_t add\_vec3d(const point3d\_t x, const point3d\_t y) {

const point3d\_t vec = { x.x[0] + y.x[0], x.x[1] + y.x[1], x.x[2] + y.x[2] };

return vec;

}

inline point3d\_t scalarmul\_vec3d(const double alpha, const point3d\_t x) {

const point3d\_t vec = { alpha \* x.x[0], alpha \* x.x[1], alpha \* x.x[2] };

return vec;

}

inline point3d\_t mat\_vec3d(const double alpha, const mat3d\_t mat, const point3d\_t x) {

const point3d\_t vec = { alpha \* (mat.a[0] \* x.x[0] + mat.a[1] \* x.x[1] + mat.a[2] \* x.x[2]),

alpha \* (mat.a[3] \* x.x[0] + mat.a[4] \* x.x[1] + mat.a[5] \* x.x[2]),

alpha \* (mat.a[6] \* x.x[0] + mat.a[7] \* x.x[1] + mat.a[8] \* x.x[2]) };

return vec;

}

inline point3d\_t grad\_f\_ext(const point3d\_t x, const double rk) {

const double g1\_x = g1\_cut(x);

const double g2\_x = g2\_cut(x);

point3d\_t grad;

const double tmp = g2\_x > 0.0 ? 2.0 \* x.x[2] : 0.0;

if (g1\_x > 0.0) {

grad.x[0] = 2.0 \* (x.x[0] - 1.0) + rk \* 2.0 \* x.x[0] \* g1\_x;

grad.x[1] = 2.0 \* (x.x[1] - 1.0) + rk \* 2.0 \* x.x[1] \* g1\_x;

grad.x[2] = 2.0 \* (x.x[2] - 1.0) + rk \* (x.x[2] \* 2.0 \* g1\_x + tmp);

}

else {

grad.x[0] = 2.0 \* (x.x[0] - 1.0);

grad.x[1] = 2.0 \* (x.x[1] - 1.0);

grad.x[2] = 2.0 \* (x.x[2] - 1.0) + rk \* tmp;

}

return grad;

}

inline point3d\_t grad\_f\_int(const point3d\_t x, const double rk) {

const double g1\_x = -g1(x);

const double inv\_g1\_x = 1.0 / g1\_x;

const point3d\_t grad = {

2.0 \* (x.x[0] \* (1.0 + rk \* inv\_g1\_x) - 1.0),

2.0 \* (x.x[1] \* (1.0 + rk \* inv\_g1\_x) - 1.0),

2.0 \* (x.x[2] - 1.0 + rk \* (x.x[2] \* inv\_g1\_x - 0.5 / x.x[2]))

};

return grad;

}

inline double phi(const double t, const point3d\_t x, const double rk, const point3d\_t d, double(\*p)(point3d\_t, double)) {

const point3d\_t xtd = add\_vec3d(x, scalarmul\_vec3d(t, d));

return f(xtd) + p(xtd, rk);

}

double methodGoldenSection(const segment\_t seg, const double epsilon, const point3d\_t x, const double rk, const point3d\_t d, double(\*p)(point3d\_t, double)) {

//printf("Start The method of the Golden section...\n");

double ak = seg.a;

double bk = seg.b;

// (3 - sqrt(5))/2 ~ 0.3819660112501051

double yk, zk, f\_yk, f\_zk;

do {

bk = bk / 2;

yk = ak + 0.3819660112501051 \* (bk - ak);

zk = ak + bk - yk;

f\_yk = phi(yk, x, rk, d, p);

f\_zk = phi(zk, x, rk, d, p);

} while (f\_yk == -(double)INFINITY || f\_zk == -(double)INFINITY);

double convergence;

uint16\_t k = 0;

do {

if (f\_yk <= f\_zk) {

bk = zk;

zk = yk;

f\_zk = f\_yk;

yk = ak + bk - yk;

f\_yk = phi(yk, x, rk, d, p);

}

else {

ak = yk;

yk = zk;

f\_yk = f\_zk;

zk = ak + bk - zk;

f\_zk = phi(zk, x, rk, d, p);

}

convergence = fabs(ak - bk);

k++;

} while (convergence > epsilon);

//printf("End The method of the Golden section\nIters = %" PRIu16 "\nConvergence = %e\n", k, convergence);

return (ak + bk) \* 0.5;

}

inline double findT(const point3d\_t x, const double rk, const point3d\_t d, double(\*p)(point3d\_t, double)) {

const segment\_t seg = { 0, 100 };

const double epsilon = 1.0e-6;

const double tmin = methodGoldenSection(seg, epsilon, x, rk, d, p);

return tmin;

}

point3d\_t nonlinearConjugateGradientMethod(const point3d\_t x0, const double epsilon, const uint32\_t maxIter, const double rk,

double(\*p)(point3d\_t, double), point3d\_t(\*grad\_fun)(point3d\_t, double)) {

//printf("Start Nonlinear Conjugate Gradient Method...\n");

const double epsilon2 = epsilon \* epsilon;

point3d\_t xk1 = x0;

point3d\_t d;

bool is\_seq = false;

uint32\_t k = 0;

double dot\_grad\_xk, convergence;

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

const point3d\_t grad\_fun\_xk1 = grad\_fun(xk1, rk);

const double dot\_grad\_xk1 = dotProduct3d(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\_vec3d(-1.0, grad\_fun\_xk1);

}

else {

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

d = add\_vec3d(scalarmul\_vec3d(beta, d), scalarmul\_vec3d(-1.0, grad\_fun\_xk1));

}

const point3d\_t xk = xk1;

const double t = findT(xk, rk, d, p);

const point3d\_t xk1\_minus\_xk = scalarmul\_vec3d(t, d);

xk1 = add\_vec3d(xk1, xk1\_minus\_xk);

convergence = dotProduct3d(xk1\_minus\_xk, xk1\_minus\_xk);

const double abs\_fk1\_minus\_fk = fabs(p(xk1, rk) - p(xk, rk));

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));

//printf("End Nonlinear Conjugate Gradient Method.\n");

return xk1;

}

#endif MAIN\_H

**main.c**

#include "main.h"

point3d\_t methodExternalPenalty(point3d\_t x0, double r0, double C, double epsilon);

point3d\_t methodInternalPenalty(point3d\_t x0, double r0, double C, double epsilon);

int8\_t main() {

const point3d\_t x0 = { 1, 1, 1 };

const double r0 = 0.1;

const double C = 2.0;

const double epsilon = 1e-6;

point3d\_t xmin;

const uint8\_t test = 1; // 1 - External Penalty metod

// 2 - Internal Penalty metod

switch (test) {

case 1:

xmin = methodExternalPenalty(x0, r0, C, epsilon);

break;

case 2:

xmin = methodInternalPenalty(x0, r0, C, epsilon);

break;

default:

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

system("pause");

return -1;

}

printf("\nf((%.2f, %.2f, %.2f)) = %.2f\n", xmin.x[0], xmin.x[1], xmin.x[2], f(xmin));

system("pause");

return 0;

}

point3d\_t methodExternalPenalty(const point3d\_t x0, const double r0, const double C, const double epsilon) {

printf("Start External Penalty metod...\n");

printf("x0 = (%.2f, %.2f, %.2f)\n", x0.x[0], x0.x[1], x0.x[2]);

printf("r0 = %f; C = %f; epsilon = %.2e\n", r0, C, epsilon);

const uint32\_t maxIter = 10000;

double rk = r0;

point3d\_t xk = x0;

double pk = 0.0;

uint32\_t k = 0;

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

xk = nonlinearConjugateGradientMethod(xk, epsilon, maxIter, rk, p\_ext, grad\_f\_ext);

pk = p\_ext(xk, rk);

if (pk <= epsilon) {

break;

}

rk \*= C;

}

printf("Iters = %" PRIu32 "\nP(x, rk) = %.2e\n", k, pk);

printf("End External Penalty metod.\n");

return xk;

}

point3d\_t methodInternalPenalty(const point3d\_t x0, const double r0, const double C, const double epsilon) {

printf("Start Internal Penalty metod...\n");

printf("x0 = (%.2f, %.2f, %.2f)\n", x0.x[0], x0.x[1], x0.x[2]);

printf("r0 = %f; C = %f; epsilon = %.2e\n", r0, C, epsilon);

const uint32\_t maxIter = 100;

double rk = r0;

point3d\_t xk = x0;

double pk = 0.0;

uint32\_t k = 0;

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

xk = nonlinearConjugateGradientMethod(xk, epsilon, maxIter, rk, p\_int, grad\_f\_int);

pk = p\_int(xk, rk);

if (fabs(pk) <= epsilon) {

break;

}

rk /= C;

}

printf("Iters = %" PRIu32 "\nP(x, rk) = %.2e\n", k, pk);

printf("End Internal Penalty metod.\n");

return xk;

}