МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра теоретической и прикладной информатики

Отчёт по учебной практике: практики по получению первичных профессиональных умений и навыков

с 01.09.2016 по 31.12.2016

|  |  |
| --- | --- |
| Студент: | Мамыкин К.А. |
|  |  |
| Группа: | ПМ-33 |
|  |  |
| Место практики: | НГТУ, кафедра ТПИ |
|  |  |
| Руководитель практики: | Лисицин Д.В. |

# Введение:

Для поиска минимума/максимума функционала существует большое количество различных методов. В данной работе будет реализован метод Бройдена-Флетчера-Гольдфарба-Шанно (БФГШ). Это метод второго порядка, что позволяет ему хорошо сходиться к локальным экстремумам. БФГШ принадлежит к классу Квазиньютоновских методов.

# Цель работы:

Реализовать алгоритм БФГШ для поиска локального экстремума функционала без учета ограничений в оптимизационной задаче.

# Математическая постановка задачи:

Запишем оптимизационную задачу: 

Разложим функцию в ряд Тейлора, отбросив члены, у которых порядок выше третьего:   
 - Матрица вторых частных производных, или матрица Гессе ().

В выражении проведем замену. Воспользуемся необходимым условием экстремума и выразим приращение:



Отсюда получаем: . Поиск обратной матрицы это вычислительно затратный процесс, поэтому вводится аппроксимация матрицы Гессе:

,

где , .  
Предполагается, что матрица п.о. матрица.

В таком виде может наблюдаться негативное влияние на метод начальной точки. Поэтому было введено понятие «регулировки шага»(параметр ).  
, где , вектор направления убывания функционала.

Это, в свою очередь, приводит к проблеме выбора шага, которая решается с помощью метода одномерного поиска. В данной работе, это правило Вулфа, в котором вводятся следующие два условия:

 ()

 ()

# Алгоритмы:

## Алгоритм БФГШ:

Инициализация, где  - заданная точность,  - заданная точка,  - размерность задачи.

Пока  и 

Поиск , k=0

Поиск по правилу Вулфа



Если = n или не п.о.

То 

Иначе вычислить , ,



## Алгоритм правила Вулфа:

Инициализация  , , где 

1. Проверяем выполнение неравенств (1) и (2). Если оба выполнены, то переходим к п.6.
2. Если нарушено (1), то полагаем  и переходим к п.5.
3. Если нарушено (2), то полагаем .
4. Если , то выбираем новое пробное значение («экстраполяция») и переходим к п.1.
5. Выбираем новое пробное значение («интерполяция») и переходим к п.1.
6. Полагаем .

# Краткое описание структур программы: class vect – класс вектор, которая содержит размерность, вещественный массив и перегруженные операторы с данной структурой.

class matrix – классматрица, с аналогичным назначением.

class QNM – класс Квазиньютоновский метод. Содержит реализацию алгоритмов БФГШ, правила Вулфа(QNM\_solution и QNM::Wulf соответственно).

double f\_x – функция, с заданным аналитически функционалом оптимизационной задачи. Возвращает значение в точке.

vect d\_f – функция, с заданной аналитически производной функционала. Возвращает градиент.

# Тестирование:



Функция: 

Минимум: (1.000000, 1.000000)

Результат работы программы : (1,1)



Функция: 

Минимум: (5, 6)

Результат работы программы : (4.99747 , 5.99592)



Функция: 

Минимум: 

Результат работы программы : (-0.547198, -1.5472)



Функция: 

Минимум: нет глобального минимума.

Результат работы программы : выход по итерациям, (-inf, -inf)



Функция: 

Минимум: (,)

Результат работы программы : (0.000244241, 0.000244241)

# Вывод:

На тестах реализованный алгоритм БФГШ выполняет поиск локального минимума функционала с заданной точностью.

# Текст программы:

**<main.cpp>**  
#include "Q.h"

void main()

{

vect result;

QNM Q("input.txt");

result = Q.QNM\_solution();

\_getch();

}

**<Q.h>**

#include <iostream>

#include <fstream>

#include <stdio.h>

#include <Windows.h>

#include <conio.h>

using namespace std;

class vect

{

private:

double \*v;

int dim;

public:

inline vect()

{

v = NULL;

}

inline vect(int dimention)

{

v = new double[dimention];

dim = dimention;

}

inline double &operator [] (unsigned i)

{

return v[i];

}

inline vect operator + (vect x)

{

vect res(dim);

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

{

res[i] = v[i] + x[i];

}

return res;

}

inline vect operator - (vect x)

{

vect res(dim);

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

{

res[i] = v[i] - x[i];

}

return res;

}

inline vect operator = (double \*res)

{

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

{

v[i] = res[i];

}

return \*this;

}

inline vect operator \* (double c)

{

vect res(dim);

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

{

res[i]=v[i] \*c;

}

return res;

}

inline vect operator / (double c)

{

vect res(dim);

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

{

res[i]=v[i] /c;

}

return res;

}

inline double\*\* operator \* (vect b)//v\*b; v - column; b - row;

{

double \*\*res;

res = new double\*[dim];

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

{

res[i] = new double [dim];

for(int j = 0; j < dim; j++)

{

res[i][j] = v[i]\*b[j];

}

}

return res;

}

double scal\_mul(vect b)

{

double res=0;

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

{

res += v[i] \* b[i];

}

return res;

}

double check\_norm()

{

double res = 0;

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

{

res += pow(v[i],2);

}

return sqrt(res);

}

};

class matrix

{

public:

inline matrix(int dimention)

{

dim = dimention;

M = new vect\*[dim];

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

{

M[i] = new vect(dim);

}

}

inline vect & matrix::operator [] (unsigned j)

{

return \*M[j];

}

inline matrix operator = (double \*\*X)

{

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

{

for (int j = 0; j < dim; j++)

{

\*M[i] = X[i];

}

}

return \*this;

}

inline vect operator \*(vect x)//A\*b

{

double line\_sum = 0;

vect res(3);

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

{

line\_sum = 0;

for (int j = 0; j < dim; j++)

{

line\_sum += (\*M[i])[j] \* x[j];

}

res[i] = line\_sum;

}

return res;

}

inline matrix operator \*(double x)//A\*scal\_value

{

matrix res(dim);

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

{

res[i] = \*M[i] \* x;

}

return res;

}

inline matrix operator +(matrix X)

{

matrix res(dim);

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

{

res[i] =\*M[i] + X[i];

}

return res;

}

inline matrix operator -(matrix X)

{

matrix res(dim);

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

{

res[i] = \*M[i] - X[i];

}

return res;

}

inline matrix operator /(double x)

{

matrix res(dim);

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

{

res[i] = \*M[i] / x;

}

return res;

}

matrix transpose()

{

matrix res(dim);

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

{

for (int j = 0; j < dim; j++)

{

res[i][j] = (\*M[j])[i];

}

}

return res;

}

private:

int dim;

vect \*\*M;

};

class QNM

{

public:

QNM(char \*filename)

{

//read data

fstream fin;

fin.open(filename);

fin >> dim >> qnm\_eps;

//allocate

Q = new double\*[dim];

x = new double [dim];

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

fin >> x[i];

fin.close();

//Q is being initialised with Diag(dim) matrix

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

{

Q[i] = new double [dim];

for(int j=0; j<dim; j++)

{

Q[i][j] = 0;

}

Q[i][i] = 1;

}

}

~QNM()

{

delete x;

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

{

delete Q[i];

}

delete Q;

}

int check\_diag\_by\_LLT(matrix Q\_new);

double Wulf(vect xk, vect dk);

vect QNM\_solution();

double find\_min(vect x,vect d, double a, double b);

private:

int dim;

double \*\*Q; //Q0 matrix

double \*x; //x0

double qnm\_eps;

int max\_iter;

};

**<Q.cpp>**

#include "Q.h"

double f\_x(vect x)

{

double result = pow(x[0] - 1E-7, 2) + pow(x[1] - 1E-7, 2);

return(result);

};

vect d\_f(vect x, int dim)

{

double result = 0; // init

vect res(dim);

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

{

switch(i)

{

case 0:

{

res[i] = 2\*(x[i] - 1E-7);

break;

}

case 1:

{

res[i] = 2\*(x[i] - 1E-7);

break;

}

default:

{

res[i] = -1;

printf("\nerror in d\_f\_x\n");

}

}

}

return res;

};

double QNM::Wulf(vect xk, vect dk)

{

double eps1 = 10E-4, eps2 = 10E-3, alpha = 5, a\_bot=0, a\_top=0; //

double thet1 = 2, thet2 = 0.5;

vect xk\_adk(dim);

int stop\_index = 0;

while (stop\_index < 1E4)

{

stop\_index++;

xk\_adk = (dk\*alpha) + xk;

double temp1 = f\_x(xk\_adk) - f\_x(xk) - eps1 \* alpha \* d\_f(xk, dim).scal\_mul(dk); // <0 true

double temp2 = d\_f(xk\_adk, dim).scal\_mul(dk) - eps2 \* d\_f(xk, dim).scal\_mul(dk); // >0 true

if (temp1 <= 0) //

{

if (temp2 >= 0)

{

break;

}

else

{

a\_bot = alpha;

if (a\_top == 0)

{

alpha \*= thet1;

}

}

}

else

{

a\_top = alpha;

alpha = (1 - thet2)\*a\_bot + thet2\*a\_top;

}

}

return alpha;

}

vect QNM::QNM\_solution()

{

fstream fout;

fout.open("output.txt");

vect d\_k(dim), s\_k(dim), r\_k(dim), x\_cur(dim), x\_new(dim), rk\_Qksk(dim);

matrix Q\_new(dim);

matrix temp(dim);

double alpha, scal\_rk\_sk = 0;

max\_iter = 0;

x\_cur = x;

x\_new = x\_cur;

Q\_new = Q;

//x\_k+1 = x\_k + a\*d\_k

int reset = 0;

for(; max\_iter < 1E+5 && d\_f(x\_new,dim).check\_norm() >= qnm\_eps ; x\_cur = x\_new, max\_iter++, reset++)

{

d\_k = d\_f(x\_cur, dim); // f'(x\_k)

d\_k = Q\_new \* d\_k;

d\_k = d\_k\*(-1); // d\_k = -Q\*f'

//Wulf

alpha = Wulf(x\_cur,d\_k);

x\_new = d\_k\*alpha + x\_cur;

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

{

printf("%f ",x\_new[i]);

}

printf("\n");

//Qk+1

if (reset == dim || !check\_diag\_by\_LLT(Q\_new))

{

Q\_new = Q;

reset = -1;

}

else

{

r\_k = x\_new - x\_cur;

s\_k = d\_f(x\_new, dim) - d\_f(x\_cur, dim);

rk\_Qksk = r\_k - Q\_new \* s\_k;

scal\_rk\_sk = r\_k.scal\_mul(s\_k);

temp = rk\_Qksk \* r\_k;

Q\_new = Q\_new + (temp.transpose() + temp) / scal\_rk\_sk;

temp = r\_k \*r\_k;

Q\_new = Q\_new - temp \* rk\_Qksk.scal\_mul(s\_k) / (scal\_rk\_sk\*scal\_rk\_sk);

}

}

printf("iterations = %d \n", max\_iter);

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

{

printf("%f ", x\_cur[i]);

fout << x\_cur[i] <<" \n";

}

return x\_cur;

}

int QNM::check\_diag\_by\_LLT(matrix Q\_new)

{

double sum1, sum2;

matrix LLT(dim);

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

{

sum1 = 0;

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

{

sum2 = 0;

for (int k = 0; k<j; k++)

{

sum2 += LLT[i][k] \* LLT[j][k];

}

LLT[i][j] = (Q\_new[i][j] - sum2) / LLT[j][j];

sum1 += LLT[i][j] \* LLT[i][j];

}

LLT[i][i] = sqrt(Q\_new[i][i] - sum1);

//check diag

if (LLT[i][i] < 0)

return 0;

}

return 1;

}