Министерство образования Республики Беларусь

Учреждение образования «Витебский государственный технологический университет»

Кафедра информационных систем и автоматизации производства

Расчетно-графическая работа

по дисциплине «Оптимизация проектных решений»

Вариант 17

Выполнил: студент группы Ит-4  
 Маслов Евгений Сергеевич

Проверил: старший преподаватель   
 кафедры ИСАП   
 Бизюк Андрей Николаевич

Витебск, 2019

# Задание 1

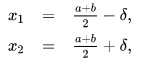
Найти точку минимума функции методами дихотомии, золотого сечения, Фибоначчи. Сравнить скорости сходимости методов.

Вариант 17 – f(x) = 4 \* ex  - 3 \* x5

## Метод дихотомии

Данный метод применяется для нахождения значений действительно-значной функции, определяемых по какому-либо критерию (это может быть сравнение на минимум, максимум или конкретное число).

Разобьём заданный отрезок пополам и возьмём две симметричные относительно центра точки *x1* и *x2* так, что:



где δ — некоторое число в интервале (0, (b-a)/2)

Вычислим два значения функции f(x) в двух новых точках. Сравнением определим в какой из двух новых точек значение функции f(x) максимально. Отбросим тот из концов изначального отрезка, к которому точка с максимальным значением функции оказалась ближе, то есть:

* Если f(x1) > f(x2)то берётся отрезок (x1,b), а отрезок (a,x1) отбрасывается.
* Иначе берётся зеркальный относительно середины отрезок (a,x2), а отбрасывается (x2,b).

Процедура повторяется, пока не будет достигнута заданная точность.

Код программы:

**public** **class** Dihot {

**int** count = 0;

**static** **double** f(**double** x) {

**return** 4 \* Math.*exp*(x) - 3 \* Math.*pow*(x, 5);

}

**double** dich(**double** a, **double** b, **double** delta, **double** eps) {

**double** x1, x2;

**if** ((b - a) > eps) {

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

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

**if** (*f*(x1) > *f*(x2)) {

count++;

**return** dich(x1, b, delta / 2, eps);

} **else** {

count++;

**return** dich(a, x2, delta / 2, eps);

}

} **else** {

count++;

**return** a;

}

}

**public** **static** **void** main(String[] args) {

Dihot d = **new** Dihot();

**double** x = d.dich(0, 20, 0.05, 0.0001);

System.***out***.println("Iterations= " + d.count + " x = " + x + " f(x) = " + *f*(x));

}

}

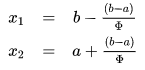


Рисунок 1 – Результат выполнения метода дихотомии

## Метод золотого сечения

Золотое сечение — это деление какой-либо величины в отношении 62% и 38%; деление отрезка АВ точкой С на две части (меньший отрезок АС и больший отрезок ВС), чтобы для длин отрезков было верно AC/BC = BC/AВ.

Для того, чтобы найти неопределённое значение этой функции на заданном отрезке, отвечающее критерию поиска, рассматриваемый отрезок делится в пропорции золотого сечения в обоих направлениях, то есть выбираются две точки *x1* и *x2* такие, что:



То есть точка *x1* делит отрезок (a,x2) в отношении золотого сечения. Аналогично x2 делит отрезок (x1,b) в той же пропорции.

Далее рассчитывают значения целевой функции в этих точках f(x1) и f(x2).

* Если f(x1) > f(x2), то a = x1;
* Иначе b = x2.

Если |b – a| < ε(точность), то x = (a+b)/2 и остановка поиска, иначе возвращаемся к рассчитыванию точек деления *x1* и *x2*.

Код программы:

**public** **class** Gold {

**final** **static** **double** ***PHI*** = (1 + Math.*sqrt*(5)) / 2;

**int** count;

**static** **double** f(**double** x) {

**return** 4 \* Math.*exp*(x) - 3 \* Math.*pow*(x, 5);

}

**double** gold(**double** a, **double** b, **double** x, **double** y, **double** n, **double** eps) {

**double** x1, x2, f1, f2;

**if** ((b - a) > eps) {

**if** (n == 1) {

x1 = 2 \* (a + (b - a) / 2) - x;

x2 = x;

f1 = *f*(x1);

f2 = y;

} **else** {

x1 = x;

x2 = 2 \* (a + (b - a) / 2) - x;

f1 = y;

f2 = *f*(x2);

}

**if** (f1 > f2) {

count++;

**return** gold(x1, b, x2, f2, 2, eps);

} **else** {

count++;

**return** gold(a, x2, x1, f1, 1, eps);

}

} **else** {

count++;

**return** a;

}

}

**public** **static** **void** main(String[] args) {

Gold GS = **new** Gold();

**double** x, x1;

x1 = (20 + 0) / ***PHI***;

x = GS.gold(0, 20, x1, *f*(x1), 1, 0.01);

System.***out***.println("Iterations=" + GS.count + " x = " + x + " f(x) = " + *f*(x));

}

}



Рисунок 2 – Результат выполнения метода золотого сечения

## Метод Фибоначчи

В этом методе количество итераций строго ограничено.

Задаются начальные границы отрезка *a* и *b*.

Далее рассчитываются начальные точки деления *x1* и *x2* и значения функции в них.

При условии n = 1, если f(x1) < f(x2), то b = x2, x2 = x1, x1 = a + (b - x2), иначе a =x1, x1 = x2, x2 = b – (x1- a)

Код программы:

**import** java.util.ArrayList;

**public** **class** Fibon {

**int** count;

**static** **double** f(**double** x) {

**return** 4 \* Math.*exp*(x) - 3 \* Math.*pow*(x, 5);

}

**public** ArrayList<Integer> F(**int** n) {

ArrayList<Integer> arr = **new** ArrayList<Integer>();

arr.add(1);

arr.add(1);

**for** (**int** i = 2; i <= n - 1; i++) {

arr.add(arr.get(i - 1) + arr.get(i - 2));

}

**return** arr;

}

**public** **double** fibonacci(**double** a, **double** b, **double** x, **double** y, **double** n, ArrayList<Integer> s, **double** l) {

**double** x1, x2, f1, f2;

**if** (count < (s.size() - 3)) {

count = count + 1;

**if** (n == 1) {

x1 = a + l \* s.get(s.size() - count - 2) / s.get(s.size() - 1);

x2 = x;

f1 = *f*(x1);

f2 = y;

} **else** {

x1 = x;

x2 = a + l \* s.get(s.size() - count - 1) / s.get(s.size() - 1);

f1 = y;

f2 = *f*(x2);

}

**if** ((f1) > (f2)) {

**return** fibonacci(x1, b, x2, f2, 2, s, l);

} **else** {

**return** fibonacci(a, x2, x1, f1, 1, s, l);

}

} **else** {

**return** a;

}

}

**public** **static** **void** main(String[] args) {

**double** x1, x;

**double** a, b;

**int** n = 18;

a = 0;

b = 20;

Fibon fbr = **new** Fibon();

ArrayList<Integer> arr;

arr = fbr.F(n);

x1 = a + (b - a) \* arr.get(n - 2) / arr.get(n - 1);

x = fbr.fibonacci(a, b, x1, *f*(x1), 1, arr, b - a);

System.***out***.println("Iterations=" + n + " x = " + x + " f(x) = " + *f*(x));

}

}



Рисунок 3 – Результат выполнения метода Фибоначчи

# Задание 2

Найти точку минимума функции методами градиентного спуска и симплексного поиска.

F(x,y)= x4 + 3y2 + 6x

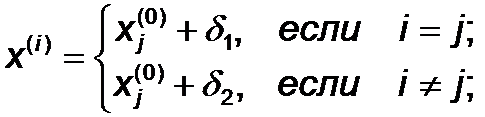
## Симплексный поиск

Суть метода заключается в исследовании целевой функции в вершинах некого "образца", построенного в пространстве вокруг "базовой" точки. Вершина, давшая наибольшее значение целевой функции отображается относительно двух других вершин и таким образом становится новой базовой точкой, вокруг которой строится новый образец и снова выполняется поиск.

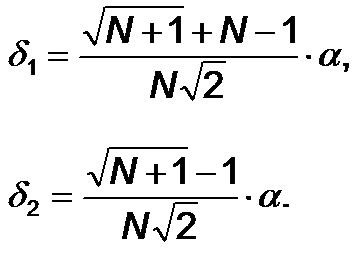
Работа алгоритма начинается с построения регулярного симплекса в пространстве независимых переменных и оценивания значений целевой функции в каждой точке. Затем определяется вершина с максимальным значением целевой функции и проектируется через центр тяжести оставшихся вершин в новую точку.

Процедура продолжается до тех пор, пока не будет накрыта точка минимума.

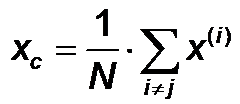
При заданной начальной точке и длине ребра, координаты остальных верщин симплекса рассчитываются по формуле:



Приращения по формулам:



Если x(i) – точка, подлежащая отражению, то координаты центра тяжести определяются по формуле:



Поиск заканчивается тогда, когда размеры симплекса или разность значений целевой функции становятся достаточно малыми.

Код программы:

public class SimplexSearch {

// public ArrayList<XY> point = new ArrayList<XY>(); //список точек

static double l = 1; // длина ребра

double n = 2; // размерность пространства (вершины)

static double delta1; // прирощения

static double delta2;

// Дельта 1

public double getDelta1() {

delta1 = (Math.sqrt(n + 1) + n - 1) \* l / (n \* Math.sqrt(2));

return delta1;

}

// Дельта 2

public double getDelta2() {

delta2 = (Math.sqrt(n + 1) - 1) \* l / (n \* Math.sqrt(2));

return delta2;

}

// Задание Функция

public static double Function(double x, double y) {

return Math.pow(x, 4) + 3 \* y\*y + 6 \* x;

}

// Находим точки первоначально симплекса

public XY[] Calculate(XY xy) {

XY[] points;

points = new XY[3];

points[0] = xy;

XY xy2 = new XY(points[0].getX() + getDelta1(), points[0].getY() + getDelta2());

points[1] = xy2;

XY xy3 = new XY(points[0].getX() + getDelta2(), points[0].getY() + getDelta1());

points[2] = xy3;

return points;

}

// поиск максимального элемента массива

int maxArr(double[] mas) {

int i;

int mx = 0;

for (i = 0; i < mas.length; i++) {

if (mas[i] > mas[mx])

mx = i;

}

return mx;

}

// поиск минимального элемента массива

int minArr(double[] mas) {

int i;

int min = 0;

for (i = 0; i < mas.length; i++) {

if (mas[i] < mas[min])

min = i;

}

return min;

}

// поиск индекса максимального

int maxIndex(XY[] points) {

double[] ptf = new double[3];

for (int i = 0; i < ptf.length; i++)

ptf[i] = Function(points[i].getX(), points[i].getY());

return maxArr(ptf);

}

// поиск индекса минимального

int minIndex(XY[] points) {

double[] ptf = new double[3];

for (int i = 0; i < ptf.length; i++)

ptf[i] = Function(points[i].getX(), points[i].getY());

return minArr(ptf);

}

// поиск центра тяжести

XY simC(XY[] points, int index) {

double сx = 0, сy = 0;

for (int i = 0; i < points.length; i++) {

if (i != index) {

сx += points[i].getX();

сy += points[i].getY();

}

}

return new XY(сx / 2, сy / 2);

}

// проверка значения функции в точке

boolean check(XY[] points, int index, XY newP) {

double v1 = Function(points[index].getX(), points[index].getY());

double v2 = Function(newP.getX(), newP.getY());

if (v2 < v1) {

points[index] = newP;

return true;

} else {

return false;

}

}

// поиск симметричной точки

public XY newPoint(XY[] points, int index) {

double x;

double y;

x = 2 \* simC(points, index).getX() - points[index].getX();

y = 2 \* simC(points, index).getY() - points[index].getY();

return new XY(x, y);

}

// реализация симплекса

public XY Search(XY xy) throws IOException {

XYSeriesCollection xyDataset = new XYSeriesCollection();

XYSeries series = new XYSeries("Graphic");

XY[] pt = Calculate(xy);

int index;

int k = 0;

series.add(pt[0].getX(), pt[0].getY());

while (true) {

index = maxIndex(pt);

XY pts = newPoint(pt, index);

if (check(pt, index, pts) == true) {

series.add(pts.getX(), pts.getY());

System.out.println(k + " " + pts.getX() + " " + pts.getY() + " " + Function(pts.getX(), pts.getY()));

} else {

series.add(pt[minIndex(pt)].getX(), pt[minIndex(pt)].getY());

xyDataset.addSeries(series);

JFreeChart chart = ChartFactory.createXYLineChart("Simplex search", "x", "y", xyDataset,

PlotOrientation.VERTICAL, true, true, true);

XYLineAndShapeRenderer renderer = new XYLineAndShapeRenderer();

final XYPlot plot = chart.getXYPlot();

renderer.setSeriesShapesVisible(0, true);

renderer.setSeriesPaint(0, Color.RED);

plot.setRenderer(renderer);

FileOutputStream fos = new FileOutputStream("123.png");

ChartUtilities.writeChartAsPNG(fos, chart, 500, 500);

return pt[minIndex(pt)];

}

k++;

}

}

public static void main(String[] args) throws IOException {

SimplexSearch sm = new SimplexSearch();

XY xy = new XY(0, 10);

xy = sm.Search(xy);

System.out

.println("MIN: X= " + xy.getX() + " Y= " + xy.getY() + " F(X,Y)= " + sm.Function(xy.getX(), xy.getY()));

}

}

//класс для хранения точки

class XY {

double x, y;

XY(double x, double y) {

this.x = x;

this.y = y;

}

public double getX() {

return x;

}

public void setX(double x) {

this.x = x;

}

public double getY() {

return y;

}

public void setY(double y) {

this.y = y;

}

}

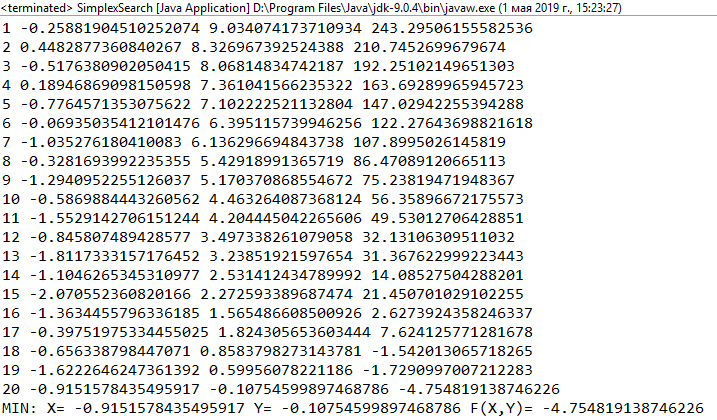


Рисунок 4 – Результат выполнения симплексного поиска

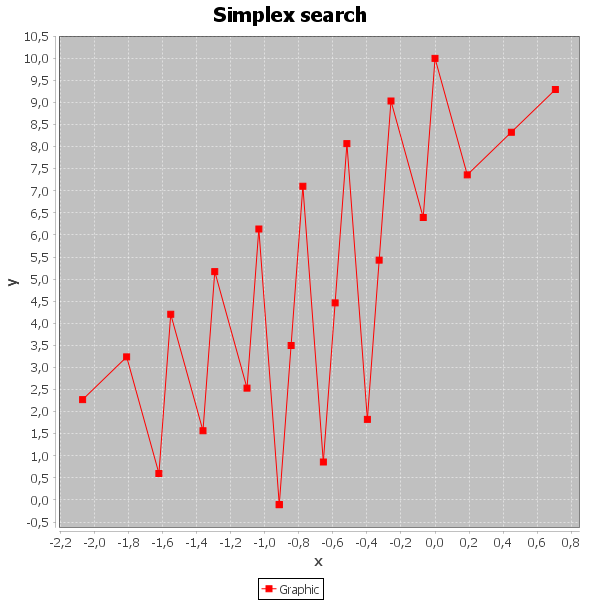
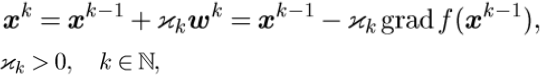


Рисунок 5 –Графическое отображение симплексного поиска

## Метод градиентного спуска

Направление спуска в методе градиентного спуска совпадает с направлением антиградиента минимизируемой целевой функции f(x), дифференцируемой в ℝ𝑛 , а элементы релаксационной последовательности {хk } строят при помощи рекуррентного соотношения



где 𝜔k = -grad f(xk-1 ) — антиградиент целевой функции в точке хk-1 . При этом говорят, что из точки хk на k-й итерации алгоритма происходит спуск с шагом спуска 𝜘𝑘 |𝜔𝑘 |. На первой итерации (k = 1) спуск начинают из выбранной начальной точки x0 .

Остановка алгоритма происходит при условии |f(xk-1)-f(xk)| > ε ,

где f(xk-1) – значение функции на предыдущем шаге,

f(xk)| – значение функции на текущем шаге, ε – точность.

Код программы:

**import** java.awt.Color;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.util.ArrayList;

**import** org.jfree.chart.ChartFactory;

**import** org.jfree.chart.ChartUtilities;

**import** org.jfree.chart.JFreeChart;

**import** org.jfree.chart.plot.PlotOrientation;

**import** org.jfree.chart.plot.XYPlot;

**import** org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;

**import** org.jfree.data.xy.XYSeries;

**import** org.jfree.data.xy.XYSeriesCollection;

**public** **class** Gradient {

//функция

**static** **double** f(**double** x, **double** y) {

**return** Math.*pow*((x + 3) \* y, 2) - 2;

}

//частная производная по х

**static** **double** dfx(**double** x, **double** y) {

**return** Math.*pow*(y, 2) \* (2 \* x + 6);

}

//частная производная по у

**static** **double** dfy(**double** x, **double** y) {

**return** Math.*pow*(x, 4) + 3 \* y\*y + 6 \* x;

}

ArrayList<XY1> gradient\_descendent(**double** x0, **double** y0, **double** eps, **double** h) {

**int** k = 0;

**double** G;

**double** Y;

**double** x1\_p;

**double** x2\_p;

ArrayList<XY1> arr = **new** ArrayList<XY1>();

**do** {

// System.out.println(x0 + " " + y0);

XY1 xy = **new** XY1();

xy.setX(x0);

xy.setY(y0);

arr.add(xy);

x1\_p = x0;

x2\_p = y0;

G = *f*(x0, y0);

x0 = x1\_p - h \* *dfx*(x1\_p, x2\_p);

y0 = x2\_p - h \* *dfy*(x1\_p, x2\_p);

Y = *f*(x0, y0);

k++;

} **while** ((Math.*abs*(Y - G)) > eps);

**return** arr;

}

**public** **static** **void** main(String[] args) **throws** IOException {

Gradient gr = **new** Gradient();

XYSeriesCollection xyDataset = **new** XYSeriesCollection();

XYSeries series = **new** XYSeries("Grad");

XY1 xy = **new** XY1();

ArrayList<XY1> arr = **new** ArrayList<XY1>();

arr = gr.gradient\_descendent(0, 10, 0.01, 0.001);

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

xy = arr.get(i);

series.add(xy.getX(), xy.getY());

System.***out***.println(xy.getX() + " " + xy.getY() + " " + *f*(xy.getX(), xy.getY()));

}

series.add(xy.getX(), xy.getY());

System.***out***.println("MIN: X= " + xy.getX() + " Y= " + xy.getY() + " F(X,Y)= " + *f*(xy.getX(), xy.getY()));

xyDataset.addSeries(series);

JFreeChart chart = ChartFactory.*createXYLineChart*("Gradient", "x", "y", xyDataset, PlotOrientation.***VERTICAL***,

**true**, **true**, **true**);

XYLineAndShapeRenderer renderer = **new** XYLineAndShapeRenderer();

**final** XYPlot plot = chart.getXYPlot();

renderer.setSeriesShapesVisible(0, **true**);

renderer.setSeriesPaint(0, Color.***RED***);

plot.setRenderer(renderer);

FileOutputStream fos = **new** FileOutputStream("123.png");

ChartUtilities.*writeChartAsPNG*(fos, chart, 600, 400);

}

}

**class** XY1 {

**private** **double** x, y;

**public** **double** getX() {

**return** x;

}

**public** **void** setX(**double** x) {

**this**.x = x;

}

**public** **double** getY() {

**return** y;

}

**public** **void** setY(**double** y) {

**this**.y = y;

}

}

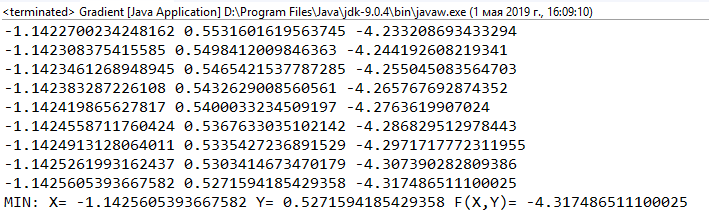


Рисунок 6 – Результат выполнения метода градиентного спуска

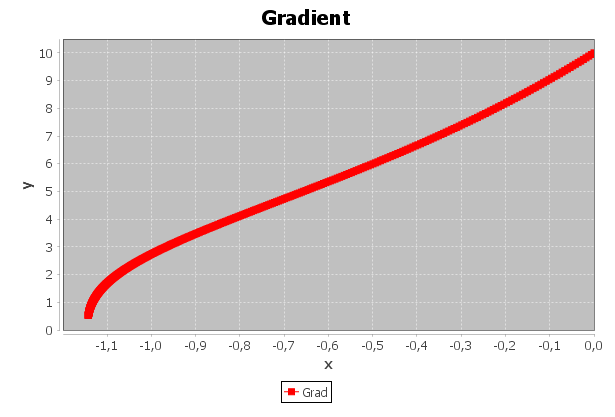


Рисунок 7 – Графическое отображения метода градиентного спуска

**Вывод:**

По методу дихотомии мы получили результат за 19 итераций, по методу золотого сечения за 17 итераций, а по методу Фибоначчи за 18 итераций. Исходя из этого метод золотого сечения быстрее остальных на промежутке от 0 до 20.