Изображение выглядит как металлоизделия, коллекция картинок

Автоматически созданное описание

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

**«Дальневосточный федеральный университет»**

**(ДВФУ)**

Институт математики и компьютерных технологий

Департамент математического и

компьютерного моделирования

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Методы оптимизации»

Выполнил студент

гр. Б9119-02.03.01сцт

Петров С.Д.

«23» марта 2022 г.

Проверил

\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«23» марта 2022 г.

г. Владивосток

2022

**Постановка задачи**

Требуется найти минимум функции нескольких переменных при помощи методов Ньютона и градиентного спуска и сравнить результаты и время работы.

**Генерация матрицы и правой части**

Для начала нужно сгенерировать симметричную и положительно определенную матрицу А. Сгенерировав нижнюю треугольную матрицу L, где все элементы на главной диагонали строго больше нуля, и умножив ее на транспонированную матрицу L мы получим необходимую симметрическую и положительно определенную матрицу. В правую часть можно записать любые числа.

**Реализация алгоритма**

Для генерации воспользуемся языком программирования Python и библиотекой Numpy для удобной работы с матрицами.

**def** **pre\_generate**():

l = np.zeros((n,n))

**for** i **in** range(n):

l[i][i] = random.randint(**1**, **100**)

**for** i **in** range(**1**,n):

**for** j **in** range(i):

l[i][j] = random.randint(-**100**, **100**)

b = np.array([random.randint(-**100**, **100**) **for** i **in** range(n)])

a = np.dot(l, np.transpose(l))

**return** [a, b]

**Нахождение первой и второй производной функции**

В обоих методах необходима производная функции, так же в методе Ньютона используется вторая производная.

Найдем их:

**Нахождение обратной матрицы**

Для метода Ньютона необходимо нахождение обратной матрицы. Будем искать обратную матрицу с помощью матрицы из алгебраических дополнений.

**Реализация алгоритма**

**def** **det**(a):

**if**(type(a[**0**]) != list **or** len(a) != len(a[**0**])):

**return** None

**if**(len(a) == **2**):

**return** a[**0**][**0**] \* a[**1**][**1**] - a[**0**][**1**]\*a[**1**][**0**]

**elif**(len(a) == **3**):

sum = **0**

**for** i **in** range(**3**):

p = **1**

m = **1**

**for** j **in** range(**3**):

p \*= a[(i + j)%**3**][j]

m \*= a[(i + j)%**3**][**2**-j]

sum += p

sum -= m

**return** sum

**else**:

sum = **0**

**for** i **in** range(len(a)):

sum += (-**1**)\*\*i \* a[**0**][i]\*det(minor(a,i,**0**))

**return** sum

**def** **minor**(a,x,y):

to\_ret = []

**for** r **in** a:

temp = r[:]

temp.pop(x)

to\_ret += [temp]

to\_ret.pop(y)

**return** to\_ret

**def** **inv**(a):

c = [r[:] **for** r **in** a]

**for** i **in** range(len(a)):

**for** j **in** range(len(a[**0**])):

c[i][j] = (-**1**)\*\*(i + j)\*det(minor(a,j,i))

**return** mul(transpose(c),**1**/det(a))

**Метод Ньютона**

Поиск значений *x* для нахождения минимума функции осуществляется по следующей формуле:

В нашем случае начальные значения *x* не влияют на ход работы и для вычисления значений *x* потребуется лишь одна итерация, потому как:

Тогда подставив в производную найденные значения получим, что мы нашли стационарную точку, являющуюся минимумом функции:

Вычисления происходили бы не за один шаг, если бы у нас была другая функция, соответственно другие производные.

**Реализация алгоритма**

Для реализации воспользуемся языком программирования Python и библиотекой Numpy для удобной работы с матрицами.

**def** **newtons\_method**(x,a,b):

**return** x - dot(inv(a), dfunc(x,a,b))

**Метод градиентного спуска**

Поиск значений *x* для нахождения минимума функции осуществляется по следующей формуле:

Будем брать , а вычисления проводить до тех пор, пока разница между значениями функций и будет больше, чем .

**Реализация алгоритма**

Для реализации воспользуемся языком программирования Python и библиотекой Numpy для удобной работы с матрицами.

**def** **grad\_method**(x,a,b,step = **0.0001**):

xn = [r[:] **for** r **in** x]

**while**(step > **0.000000001**):

start = func(xn,a,b)

d = dfunc(xn,a,b)

right = mul(d ,-step)

xn = plus(xn, right)

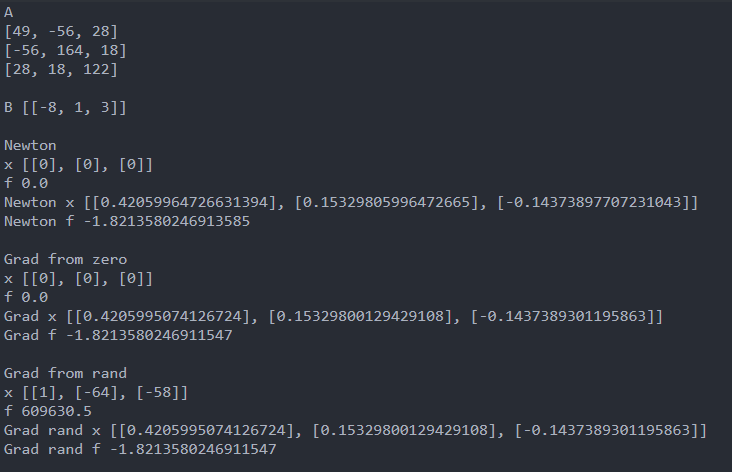
end = func(xn,a,b)

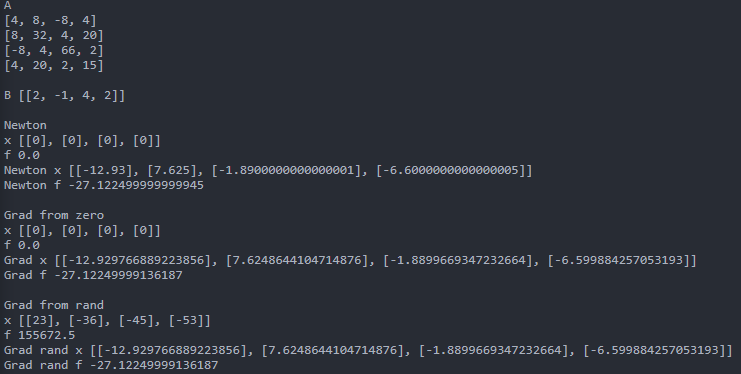
**if**(start < end):

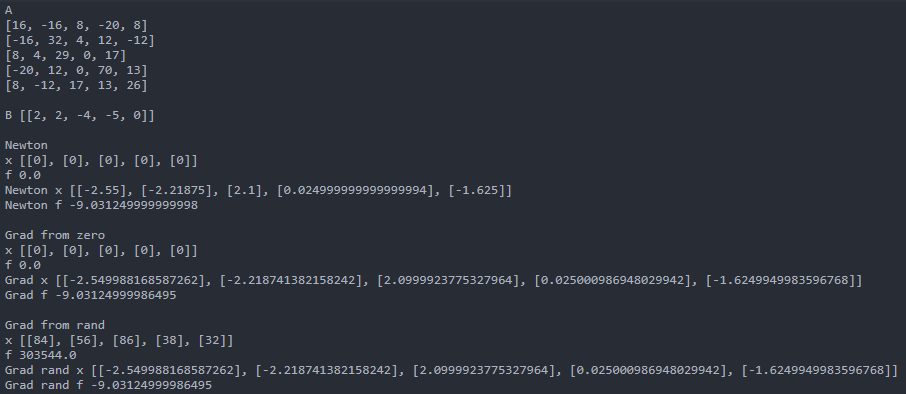
step /= **10**

**return** xn

**Тесты**

****

****

****

**Заключение**

Таким образом в данной лабораторной работе я реализовал и изучил метод Ньютона и градиентного спуска для поиска минимального значения функции нескольких переменных и провел тесты работы методов, из которых видно, что:

1. Размер матрицы сильно влияет на время работы метода градиентного спуска
2. Начальные данные влияют на скорость работы и количество итераций метода градиентного спуска