МИНОБРНАУКИ РОССИИ
федеральное государственное бюджетное образовательное
учреждение высшего образования
«Национальный исследовательский университет «МЭИ»

ЛАБОРАТОРНАЯ РАБОТА №2
По курсу: «Методы решения задач оптимизации»
Тема: «Градиентный метод»

Выполнил: Волков М.Л.
Вариант: 1
Группа: Э-13м-23
Проверил: Нухулов С.М.

Москва, 2024 г.

Предварительный отчет

Цель: получение практических навыков работы с методом решения задач нелинейного программирования с количеством неизвестных больше трех.

Задание:

1. Реализовать классический градиентный метод решения задачи выпуклого программирования;
2. Реализовать градиентный метод Momentum решения задачи выпуклого программирования;
3. Реализовать градиентный метод NAG (ускоренный градиентный метод Нестерова) решения задачи выпуклого программирования;
4. Реализовать градиентный метод RMSProp решения задачи выпуклого программирования;
5. Реализовать градиентный метод AdaDelta решения задачи выпуклого программирования;
6. Реализовать градиентный метод Adam решения задачи выпуклого программирования;
7. Построение диаграммы «ящик с усами»;

Теоретическая справка:

Изучение методов решения нелинейных задач является предметом раздела математического программирования, получившего название нелинейного программирования.

Задачи нелинейного программирования обладают следующими свойствами, которые существенно усложняют процесс их решения по сравнению с задачами ЛП:

- область допустимых решений может иметь очень сложную структуру;
- точки экстремума в задачах с нелинейной целевой функцией могут лежать как внутри области, так и на ее границе, причем локальных экстремумом может быть несколько;
- целевая функция может быть недифференцируемой, что затрудняет применение классических методов математического анализа;

То для ее решения могут быть применены классические методы, в частности, метод неопределённых множителей Лагранжа.

Порядок решения задачи методом множителей Лагранжа:

1. составить функцию Лагранжа;
2. найти частные производные функции Лагранжа по всем переменным и приравнять их нулю. Тем самым будет получена система, состоящая из $m + n$ уравнений. Решить полученную системы (если это возможно) и найти таким образом все стационарные точки функции Лагранжа;
3. из стационарных точек, взятых без координат, выбрать точки, в которых функция имеет локальные экстремумы при наличии ограничений. Этот выбор осуществляется, например, с применением достаточных условий локального экстремума;

Градиентный метод:

Ведущее место среди методов решения экстремальных задач занимают градиентные методы поиска стационарных точек дифференцируемой функции, т. е. точек, в которых частные производные обращаются в нуль. Используя эти методы, находят множество точек локальных экстремумов, среди которых определяют глобальный экстремум. Наиболее эффективны градиентные методы при решении задач выпуклого программирования, в которых всякий локальный экстремум является одновременно и глобальным.

Градиентные методы основаны на простой идее о том, что градиент функции указывает направление ее наискорейшего возрастания в окрестности той точки, в которой он вычислен. Поэтому, если известно, что функция $f(X)$ имеет в области допустимых решений единственный экстремум, то поиск точки, в которой он достигается, целесообразно организовать следующим образом:

1. выбрать произвольную точку X_0 в области допустимых решений;
2. вычислить в точке X_0 градиент функции $\nabla f(X_0)$ (или антиградиент), тем самым определить направление, в котором функция $f(X)$ возрастает (убывает) с наибольшей скоростью;
3. сделав небольшой шаг в найденном направлении, перейти в новую точку X_1. Величина шага из точки X_0 в направлении $\nabla f(X_0)$ определяется значением параметра λ в уравнении прямой $X_1 = X_0 + \lambda \cdot \nabla f(X_0)$;
4. в точке X_1 вновь определить наилучшее направление для перехода в очередную точку X_2 и т.д;
5. в результате получается последовательность точек X_0, X_1, X_2, ..., для которых выполняются условия $f(X_0) < f(X_1) < f(X_2) < \cdots$

Процесс продолжается до тех пор, пока не будет достигнута точка $\dot{X}$, в которой $\nabla f(\dot{X}) = 0$.

Рис. 1 – Алгоритм градиентного метода

В данной лабораторной работе будут рассматриваться следующие виды градиентного метода:

- Классический;
- Momentum;
- NAG;
- RMSProp;
- AdaDelta;
- Adam;

Градиентный метод Momentum называется также методом накопления импульса.

Градиентный метод NAG – стохастический градиент с импульсом Нестерова считает градиент в той точке, в которую мы перешли, используя вектор накопления импульса. Этот алгоритм показывает более быструю сходимость.

Градиентный метод RMSProp (running means square propagation) – метод с адаптацией скорости изменения весов, скользящим средним. Веса с течением времени могут меняться, если было выбрано неправильное направление.

Градиентный метод AdaDelta – метод двойно нормировки приращения весов.

Градиентный метод Adam – этот метод включает в себя комбинацию метода импульса и RMSProp.

Расчет:

Исходная функция:

$$\min_{w} f(W) = 2 \cdot w_1^2 + 2 \cdot w_2^2 - 4 \cdot w_1 - 8 \cdot w_2$$

Графическое решение нелинейной задачи с помощью графического калькулятора Desmos:

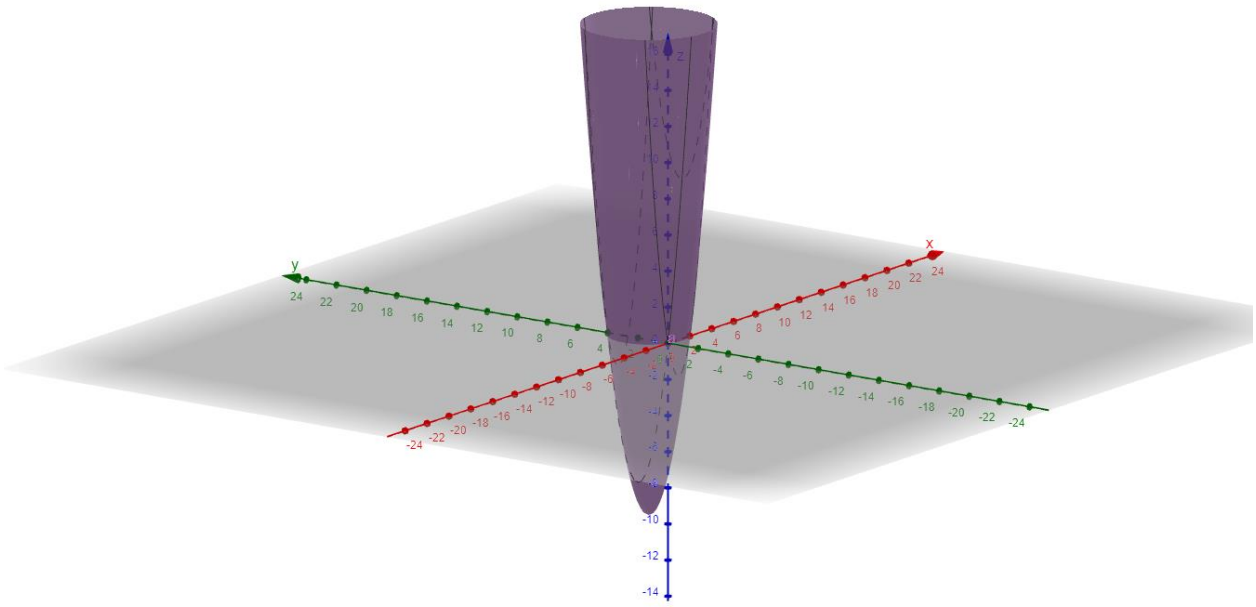

Рис. 2 – Вид с боку

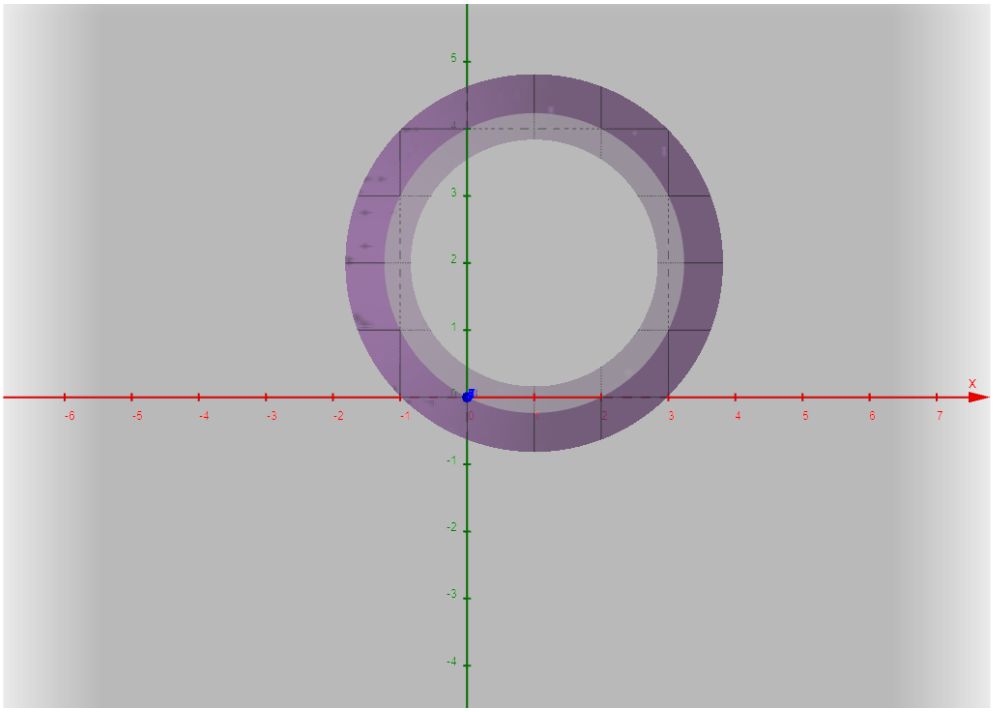

Рис. 3 – Вид сверху

Решение по методу Лагранжа:

Нахождение частных производных:

$$x = w_1' = 4 \cdot w_1 - 4 = 0$$

$$y = w_2' = 4 \cdot w_2 - 8 = 0$$

$$w_1 = 1$$
$$w_2 = 2$$

Искомая точка имеет координаты Z[1;2].

Математический расчет совпадает с графическим представлением функции.

Отчет

Написание алгоритма симплекс-метода:

Ссылка на репозиторий: https://github.com/Aglomiras/LR2_Optimize

Код:

```
import math
import random
import matplotlib.pyplot as plt

'''Инициализация общих констант'''
max_iter = 10000  # предельное количество итераций
epsilon = 1 * math.pow(math.e, -6) # точность расчета
rate = 0.01  # скорость спуска

lambda_val = 0.1  # коэффициент забывания
alpha = 0.999

'''Инициализация вспомогательных констант'''
gamma = 1 - lambda_val
eta = (1 - gamma) * rate


'''Возвращает вектор искомых значений функции. Выводит число итераций или
отсутствие решений'''
def print_message(mass, count, flag):
    if flag:
        print("Число итераций = {:d}".format(count))
        print(mass)
        return mass
    else:
        print("Решение не найдено")


def grad_descent():
    w = []
    '''заполнение начального вектора случайными числами'''
```

```
    for i in range(2):
        w.append(random.randint(-100, 100))

    Grad_W = [0, 0]  # вектор частных производных
    W1 = [0, 0]  # вектор новых значений координат точки

    '''взятие частных производных и проверка условия минимума'''
    count_flag = 0
    flag = False
    while (count_flag < max_iter):
        Grad_W[0] = (4 * w[0] - 4)
        Grad_W[1] = (4 * w[1] - 8)

        for i in range(len(Grad_W)):
            W1[i] = (w[i] - rate * Grad_W[i])

        if abs(w[0] - W1[0]) < epsilon and abs(w[1] - W1[1]) < epsilon:
            flag = True
            break
        else:
            w[0] = W1[0]
            w[1] = W1[1]

        count_flag = count_flag + 1

    print_message(W1, count_flag, flag)
    return count_flag


def grad_descent_momentum():
    w = []
    v = []
    '''заполнение начального вектора случайными числами'''
    for i in range(2):
        w.append(random.randint(-100, 100))
        v.append(random.randint(-100, 100))

    Grad_W = [0, 0]  # вектор частных производных
    W1 = [0, 0]  # вектор новых значений координат точки

    '''взятие частных производных и проверка условия минимума'''
    count_flag = 0
    flag = False
    while (count_flag < max_iter):
        Grad_W[0] = (4 * w[0] - 4)
        Grad_W[1] = (4 * w[1] - 8)
```

```
        for i in range(len(Grad_W)):
            v[i] = gamma * v[i] + eta * Grad_W[i]
            W1[i] = w[i] - v[i]

        if abs(w[0] - W1[0]) < epsilon and abs(w[1] - W1[1]) < epsilon:
            flag = True
            break
        else:
            w[0] = W1[0]
            w[1] = W1[1]

        count_flag = count_flag + 1

    print_message(W1, count_flag, flag)
    return count_flag


def grad_descent_NAG():
    w = []
    v = []
    '''заполнение начального вектора случайными числами'''
    for i in range(2):
        w.append(random.randint(-100, 100))
        v.append(random.randint(-100, 100))

    Grad_W = [0, 0]  # вектор частных производных
    W1 = [0, 0]  # вектор новых значений координат точки

    '''взятие частных производных и проверка условия минимума'''
    count_flag = 0
    flag = False
    while (count_flag < max_iter):
        Grad_W[0] = (4 * w[0] - 4) - gamma * v[0]
        Grad_W[1] = (4 * w[1] - 8) - gamma * v[1]

        for i in range(len(Grad_W)):
            v[i] = gamma * v[i] + eta * Grad_W[i]
            W1[i] = w[i] - v[i]

        if abs(w[0] - W1[0]) < epsilon and abs(w[1] - W1[1]) < epsilon:
            flag = True
            break
        else:
            w[0] = W1[0]
            w[1] = W1[1]

        count_flag = count_flag + 1
```

```
    print_message(W1, count_flag, flag)
    return count_flag


def grad_descent_RMSProp():
    w = []
    G = []
    '''заполнение начального вектора случайными числами'''
    for i in range(2):
        w.append(random.randint(-100, 100))
        G.append(0)

    Grad_W = [0, 0]  # вектор частных производных
    W1 = [0, 0]  # вектор новых значений координат точки

    '''взятие частных производных и проверка условия минимума'''
    count_flag = 0
    flag = False
    while (count_flag < max_iter):
        Grad_W[0] = (4 * w[0] - 4)
        Grad_W[1] = (4 * w[1] - 8)

        for i in range(len(Grad_W)):
            G[i] = gamma * G[i] + (1 - gamma) * Grad_W[i] * Grad_W[i]
            W1[i] = w[i] - (1 - gamma) * Grad_W[i] / math.sqrt(G[i] + epsilon)

        if abs(w[0] - W1[0]) < epsilon and abs(w[1] - W1[1]) < epsilon:
            flag = True
            break
        else:
            w[0] = W1[0]
            w[1] = W1[1]

        count_flag = count_flag + 1

    print_message(W1, count_flag, flag)
    return count_flag


def grad_descent_AdaDelta():
    Delta = 0.01  # коэффициент забывания
    delta_1 = 0.01

    w = []
    G = []
    '''заполнение начального вектора случайными числами'''
```

```
    for i in range(2):
        w.append(random.randint(-100, 100))
        G.append(0)

    Grad_W = [0, 0]  # вектор частных производных
    W1 = [0, 0]  # вектор новых значений координат точки

    '''взятие частных производных и проверка условия минимума'''
    count_flag = 0
    flag = False
    while (count_flag < max_iter):
        Grad_W[0] = (4 * w[0] - 4)
        Grad_W[1] = (4 * w[1] - 8)

        for i in range(len(Grad_W)):
            G[i] = alpha * G[i] + (1 - alpha) * Grad_W[i] * Grad_W[i]
            delta_1 = Grad_W[i] * (math.sqrt(Delta) + epsilon) / math.sqrt(G[i]
+ epsilon)
            Delta = alpha * Delta + (1 - alpha) * delta_1 * delta_1
            W1[i] = w[i] - delta_1

        if abs(w[0] - W1[0]) < epsilon and abs(w[1] - W1[1]) < epsilon:
            flag = True
            break
        else:
            w[0] = W1[0]
            w[1] = W1[1]

        count_flag = count_flag + 1

    print_message(W1, count_flag, flag)
    return count_flag


def grad_descent_Adam():
    w = []
    v = []
    G = []
    '''заполнение начального вектора случайными числами'''
    for i in range(2):
        w.append(random.randint(-100, 100))
        v.append(0)
        G.append(0)

    Grad_W = [0, 0]  # вектор частных производных
    W1 = [0, 0]  # вектор новых значений координат точки
```

```
    '''взятие частных производных и проверка условия минимума'''
    count_flag = 1
    flag = False
    while (count_flag < max_iter):
        Grad_W[0] = (4 * w[0] - 4)
        Grad_W[1] = (4 * w[1] - 8)

        for i in range(len(Grad_W)):
            v[i] = gamma * v[i] + (1 - gamma) * Grad_W[i]
            G[i] = alpha * G[i] + (1 - alpha) * Grad_W[i] * Grad_W[i]
            v_val = v[i] / (1 - math.pow(gamma, count_flag))
            g_val = G[i] / (1 - math.pow(alpha, count_flag))
            W1[i] = w[i] - rate * v_val / (math.sqrt(g_val) + epsilon)

        if abs(w[0] - W1[0]) < epsilon and abs(w[1] - W1[1]) < epsilon:
            flag = True
            break
        else:
            w[0] = W1[0]
            w[1] = W1[1]

        count_flag = count_flag + 1

    print_message(W1, count_flag, flag)
    return count_flag

grad_descent()
grad_descent_momentum()
grad_descent_NAG()
grad_descent_RMSProp()
grad_descent_AdaDelta()
grad_descent_Adam()
```

В коде представлена программная реализация всех видов градиентного метода.

Снятие диаграммы «ящик с усами» для каждого метода производится путем запуска каждого метода 100 раз, для понимания распределения количества итераций при заданных настройках.

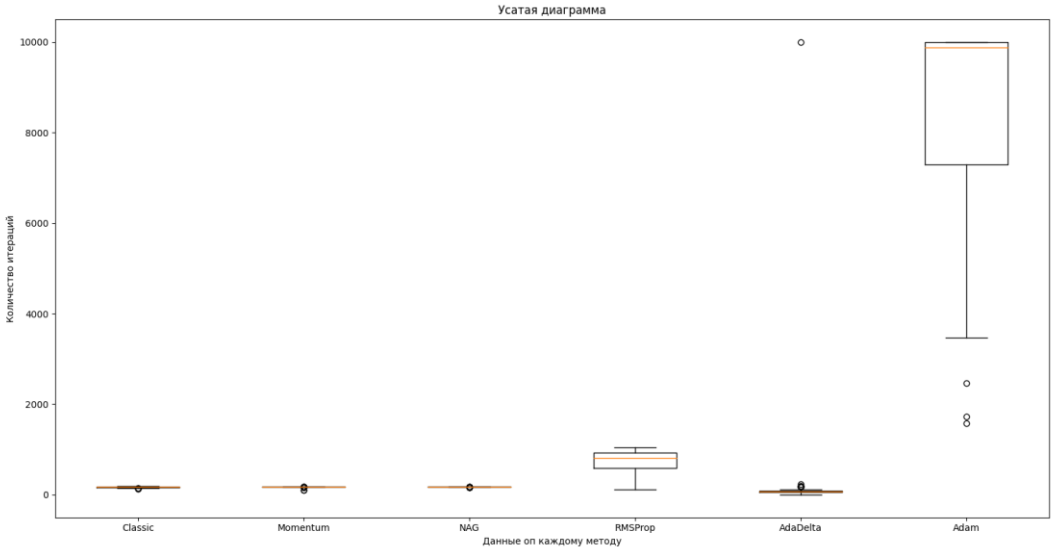

Рис. 4 – Общий вид количества итераций каждого метода

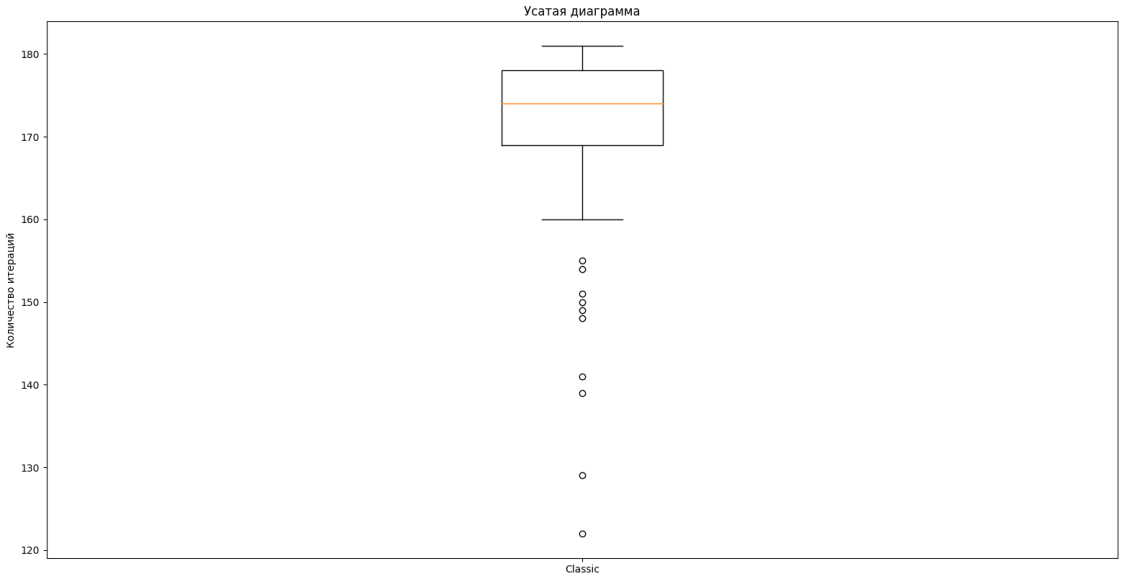

Рис. 5 – Распределение количества итераций классического метода

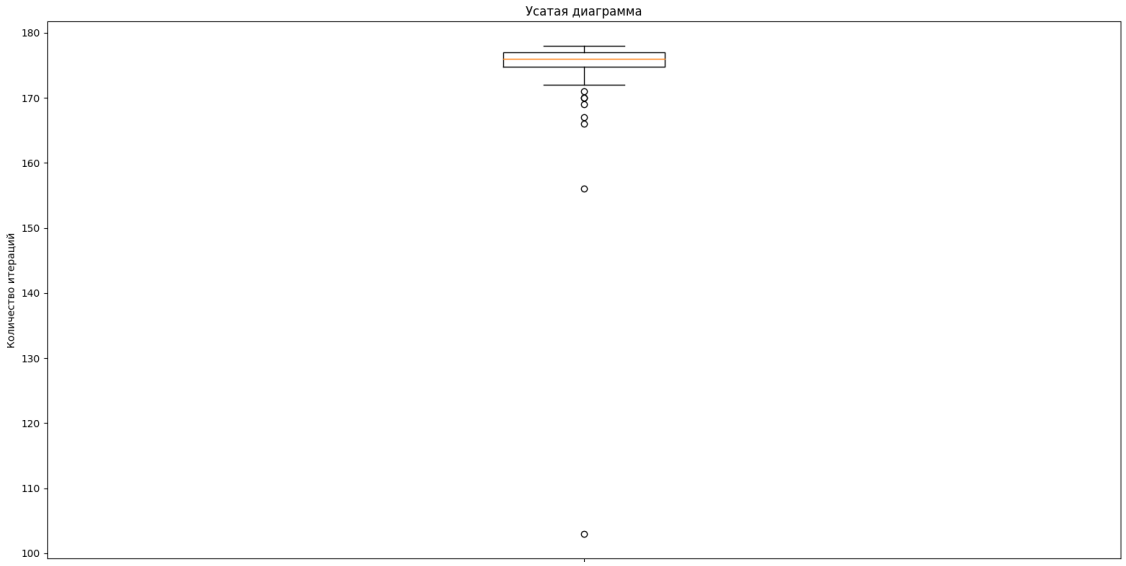

Рис. 6 – Распределение количества итераций градиентного метода Momentum

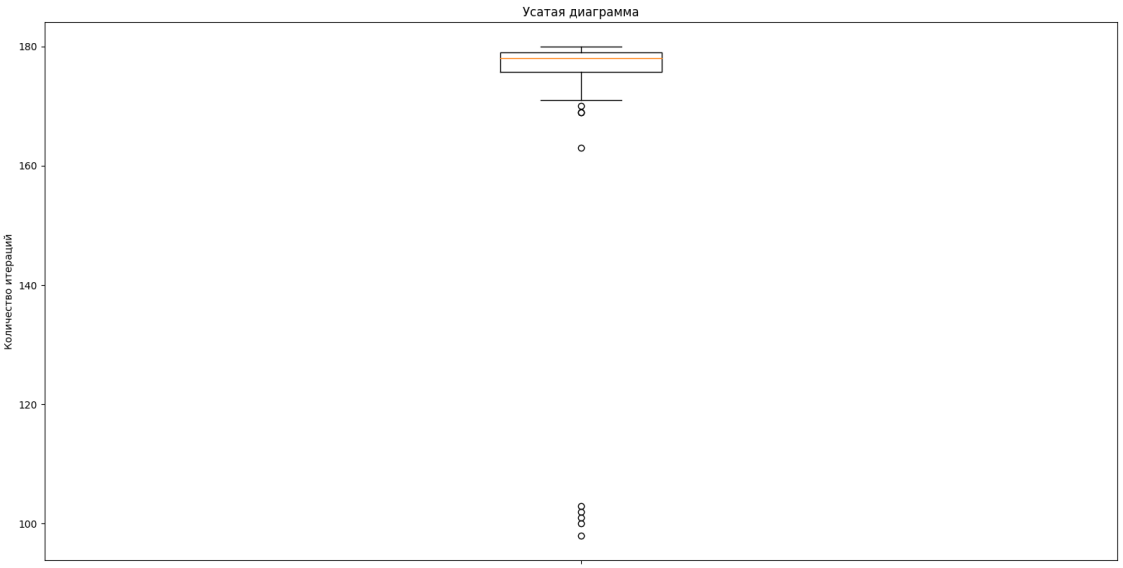

Рис.7 – Распределение количества итераций градиентного метода NGA

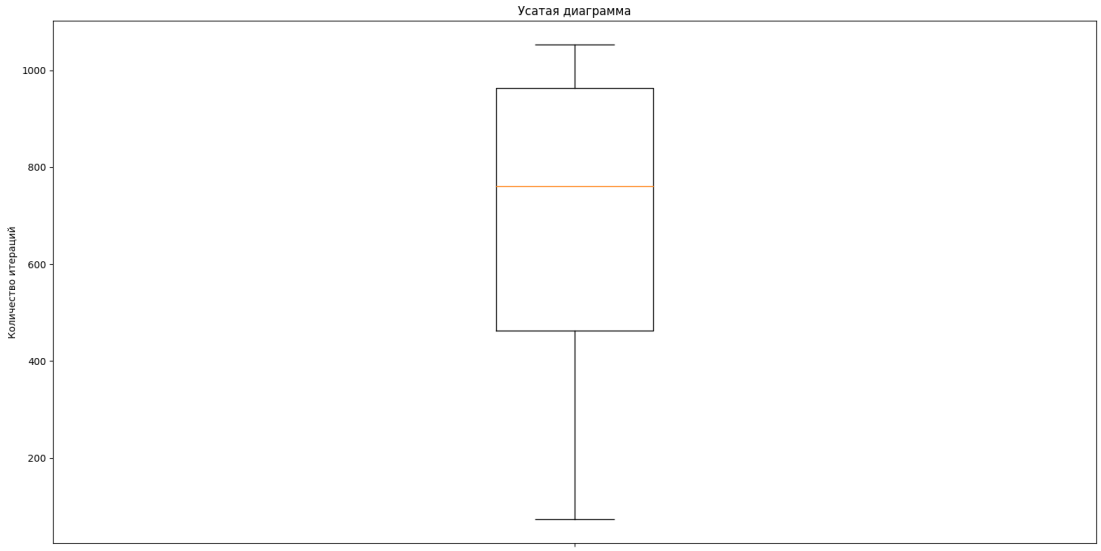

Рис. 8 – Распределение количества итераций градиентного метода RMSProp

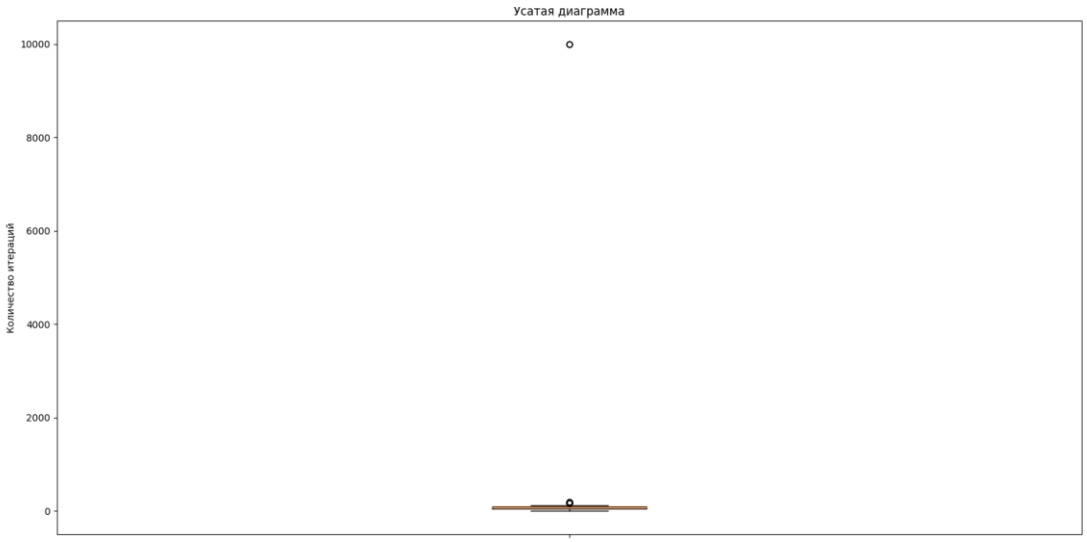

Рис. 9 – Распределение количества итераций градиентного метода AdaDelta

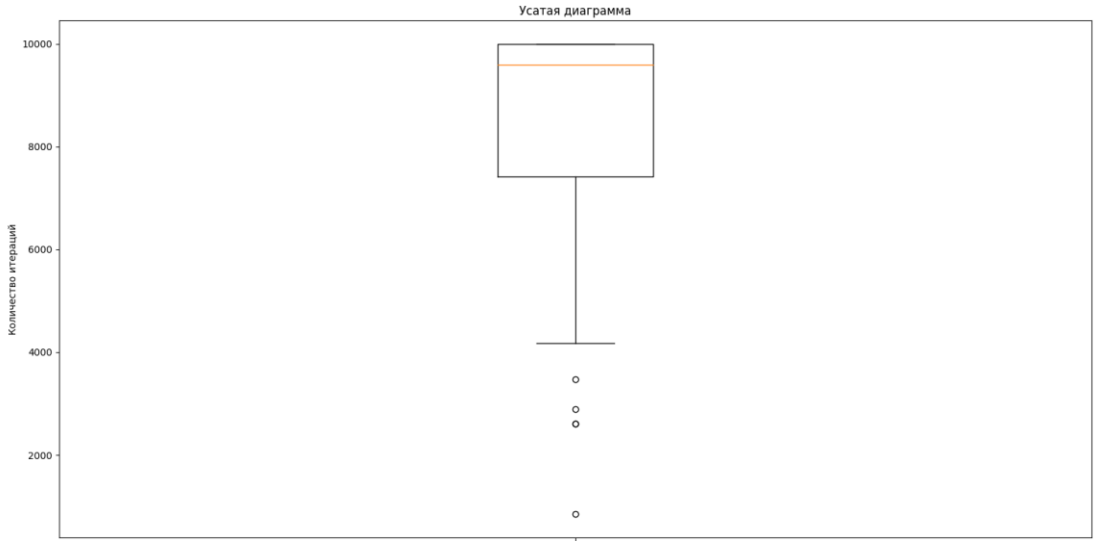

Рис. 10 – Распределение количества итераций градиентного метода Adam

Блок вывода диаграммы:

```
data = []
res = []
for i in range(100):
    res.append(grad_descent())
res1 = []
for i in range(100):
    res1.append(grad_descent_momentum())
res2 = []
for i in range(100):
    res2.append(grad_descent_NAG())
res3 = []
for i in range(100):
    res3.append(grad_descent_RMSProp())
res4 = []
for i in range(100):
    res4.append(grad_descent_AdaDelta())
res5 = []
for i in range(100):
    res5.append(grad_descent_Adam())

data.append(res)
data.append(res1)
data.append(res2)
data.append(res3)
data.append(res4)
```

```
data.append(res5)

fig, ax = plt.subplots()
bar = ['Classic', 'Momentum', 'NAG', 'RMSProp', 'AdaDelta', 'Adam']
ax.set_xlabel('Данные по каждому методу')
ax.set_xticklabels(bar)
ax.set_ylabel('Количество итераций')
ax.set_title('Усатая диаграмма')
ax.boxplot(data)
plt.show()
```

Вывод:

При проведении опытов, были заданы рекомендуемые значения скорости спуска, точности, максимального количества итераций, коэффициента забывания и коэффициента обучения.

Опытным путем было выяснено, что при уменьшении коэффициента точности, будет увеличиваться количество шагов, но при этом найденный экстремум будет намного точнее к истинному.

При увеличении коэффициента точности, ситуация будет складываться наоборот в сторону менее точного результата.

При увеличении скорости спуска, до некоторого числа, скорость нахождения экстремума будет увеличиваться, то есть количество шагов расчета сократится. Но при слишком большой скорости спуска, есть риск ухудшить ситуацию, увеличив количество шагов расчета или вовсе не найти нужного решения.

Коэффициент забывания помогает определить более точные значения координат экстремума, если его увеличить. При малых коэффициентах забывания значения могут получиться менее точными при том же числе итераций.

Ограничения в 10000 итераций хватает для первых пяти методов расчета. Для Градиентного метода Adam в большинстве случаев данного количества итераций хватать не будет, так данный метод более точный, но при этом более медленный, на его расчет требуется большее количество итераций.

Искомые координаты экстремума при запуске кода, получаются близкими, в пределах точности расчета, результатам полученным в предварительной подготовке.