МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА - Российский технологический университет»
РТУ МИРЭА

Институт Информационных Технологий
Кафедра Вычислительной техники

Отчёт по практическим работам №1-3

по дисциплине
«Математическое обеспечение СППР»

Студент группы ИКБО-04-22 Егоров Л.А.
(Ф.И.О. студента)

Принял старший преподаватель Семёнов Р.Э.
(Ф.И.О. преподавателя)

Москва 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Задача нахождения оптимального значения функции является одной из важнейших в машинном обучении, ведь большинство алгоритмов основываются на минимизации функции потерь.

Для решения задачи оптимизации могут применяться различные методы:

1. Методы нулевого порядка — позволяют решить задачу оптимизации для произвольных функций, однако обладают невысокой сходимостью.
2. Методы первого порядка — сходятся к глобальному минимуму выпуклых функций, однако для произвольных функций необходимо, чтобы в каждой точке существовала производная первого порядка. Также для произвольных функций нет гарантии, что метод сойдётся к глобальному минимуму вместо локального.
3. Методы второго порядка — аналогичны метода первого порядка, однако используют производную второго порядка и обладают более высокой сходимостью.

1 МЕТОДЫ НУЛЕВОГО ПОРЯДКА

1.1 Теоретическая информация

Методы нулевого порядка выгодны тем, что не требуют вычисления производных и что им достаточно только непрерывности целевой функции. Однако сходимость у данных методов не доказана и является эвристической.

1.2 Постановка задачи

Требуется найти точку локального минимума $\overline{x^*} = (x_1^*, x_2^*)$ целевой функции от 2-х переменных (1.1) на множестве допустимых значений.

$$2x_1^2 - 2x_1x_2 + 3x_2^2 + x_1 - 3x_2 \tag{1.1}$$

В качестве метода оптимизации используется метод Хука-Дживса, который состоит из последовательности шагов исследующего поиска относительно базисной точки и поиска по образцу.

1.3 Ручной расчёт оптимального значения функции

Перед реализацией алгоритма проведён ручной расчёт минимального значения функции, представленной Формулой 1.1. Для этого сначала проверено необходимое условие экстремума функции.

$$\begin{cases} \dfrac{\partial f}{\partial x_1} = 0 \\ \dfrac{\partial f}{\partial x_2} = 0 \end{cases} \Leftrightarrow \begin{cases} 4x_1 - 2x_2 + 1 = 0 \\ -2x_1 + 6x_2 - 3 = 0 \end{cases} \Leftrightarrow \begin{cases} x_1 = 0 \\ x_2 = 0.5 \end{cases}$$

Значение целевой функции в этой точке равно $2 \cdot 0 - 2 \cdot 0 \cdot 0.5 + 3 \cdot 0.5^2 +$ $+0 - 3 \cdot 0.5 = -0.75$.

Далее проверено достаточное условие экстремума функции. Для этого вычислены все производные второго порядка для целевой функции.

$$A = \frac{\partial^2 f}{\partial x_1^2} = 4 > 0$$

$$B = \frac{\partial^2 f}{\partial x_1 \partial x_2} = -2$$

$$C = \frac{\partial^2 f}{\partial x_2^2} = 6$$

$$AC - B^2 = 4 \cdot 6 - (-2)^2 = 28 > 0$$

Следовательно, для любых допустимых значений аргументов выполняется достаточное условие экстремума, а значит, точка $(0, 0.5)$ является точкой минимума функции. При этом, у целевой функции больше нет стационарных точек, поэтому в точка достигается глобальный минимум функции.

График функции представлен на Рисунке 1.1.

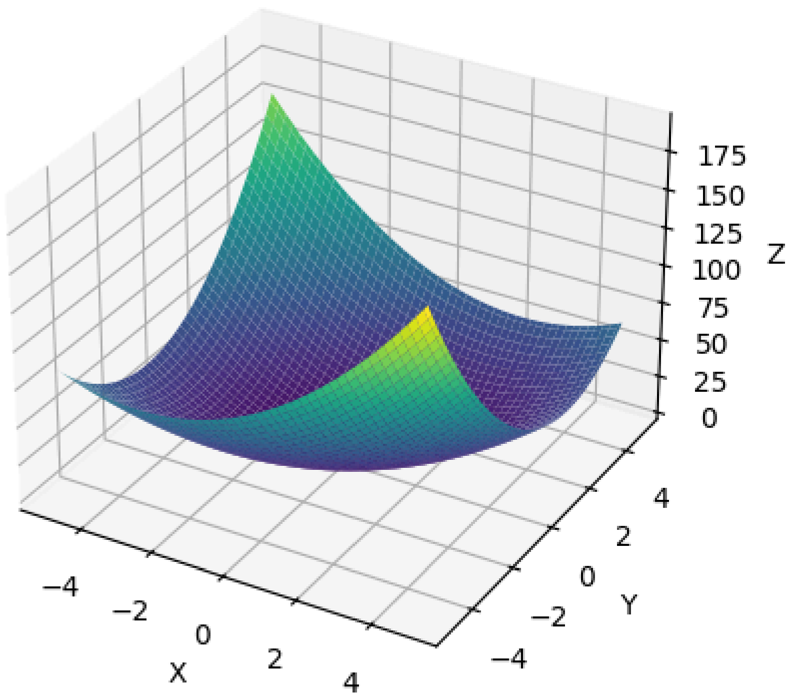

Рисунок 1.1 — График целевой функции

На Рисунке 1.2 представлен график функции в проекции на плоскость XOZ.

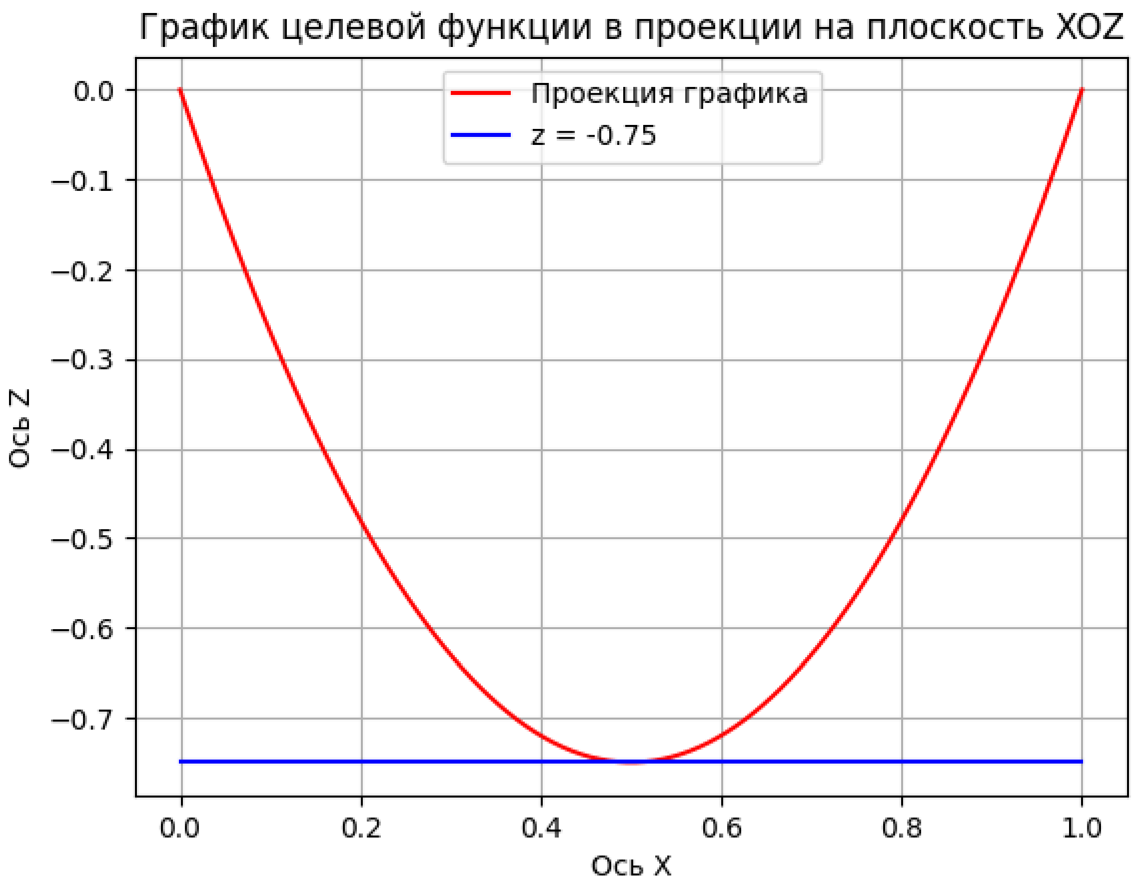

Рисунок 1.2 — График целевой функции в проекции на плоскость XOZ

1.4 Ручной расчёт алгоритма

Зафиксированы следующие параметры алгоритма:

– начальная точка: $\overline{x^{(0)}} = (1, 1)$;

– точность приближения: $\varepsilon = 0.0001$;

– коэффициент уменьшения шага: $d = 10$;

– начальная величина шага: $h = 0.2$;

– ускоряющий коэффициент: $m = 2$.

1.4.1 Первая итерация

Значение целевой функции в базисной точке: $f\left(\overline{x^{(0)}}\right) = 1$. После фиксации второй координаты $x_2^{(0)}$ и выполнения шага в положительном направлении $x_1^{(0)}$, получена точка $\overline{x^{(1)}} = \left(x_1^{(0)} + h, x_2^{(0)}\right) = (1.2, 1)$. Значение целевой функции в этой точке равно $f\left(\overline{x^{(1)}}\right) = 1.68 > f\left(\overline{x^{(0)}}\right)$, следовательно, шаг в этом направлении считается неудачным.

Далее выполнен шаг в отрицательном направлении $x_1^{(0)}$ и получена точка $\overline{x^{(1)}} = \left(x_1^{(0)} - h, x_2^{(0)}\right) = (0.8, 1)$. Значение целевой функции в этой точке равно $f\left(\overline{x^{(1)}}\right) = 0.48 < f\left(\overline{x^{(0)}}\right)$, следовательно, шаг в этом направлении считается удачным. Таким образом, фиксируется точка $\overline{x^{(1)}} = (0.8, 1)$.

После выполнения шага в положительном направлении x_2, получена точка $\left(x_1^{(1)}, x_2^{(1)} + h\right) = (0.8, 1.2)$. Значение целевой функции в этой точке равно $f(0.8, 1.2) = 0.88 > f\left(\overline{x^{(1)}}\right)$, следовательно, шаг в этом направлении считается неудачным.

Далее выполнен шаг в отрицательном направлении x_2 и получена точка $\left(x_1^{(1)}, x_2^{(0)} - h\right) = (0.8, 0.8)$. Значение целевой функции в этой точке равно $f\left(\overline{x^{(1)}}\right) = 0.32 < f\left(\overline{x^{(1)}}\right)$, следовательно, шаг в этом направлении считается удачным. Таким образом, фиксируется точка $\overline{x^{(1)}} = (0.8, 0.8)$.

Поскольку $\overline{x^{(1)}} \neq \overline{x^{(0)}}$, то выполняется поиск по образцу:

$$\overline{x^{(p)}} = \overline{x^{(1)}} + m\left(\overline{x^{(1)}} - \overline{x^{(0)}}\right) = (0.8, 0.8) + 2[(0.8, 0.8) - (1, 1)] = (0.4, 0.4)$$

$$f\left(\overline{x^{(p)}}\right) = -0.32$$

1.5 Программная реализация

На Рисунке 1.3 представлен результат работы метода Хука-Дживса. Для нахождения оптимального значения алгоритм провёл 15 итераций.

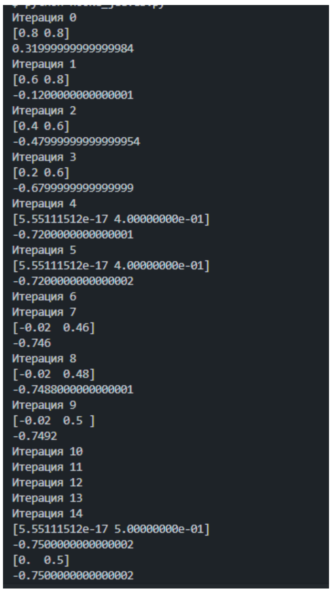

Рисунок 1.3 — Результат работы метода Хука-Дживса

Реализация метода Хука-Дживса представлена в Листинге А.1.

2 МЕТОДЫ ПЕРВОГО ПОРЯДКА

2.1 Теоретическая информация

Методы первого порядка используют производные первого порядка, что позволяют гарантировать сходимость к глобальному минимуму выпуклых функций.

Координаты новой точки в данном методе вычисляются по Формуле 2.1.

$$\overline{x^{k+1}} = \overline{x^k} - h_k \Delta f\left(\overline{x^k}\right) \tag{2.1}$$

где k — номер итерации;

h_k — величина шага. В данной работе принимается постоянной;

$\Delta f\left(\overline{x^k}\right)$ — градиент функции.

Если $f\left(\overline{x^{k+1}}\right) > f\left(\overline{x^k}\right)$, то нужно уменьшить шаг в два раза.

Условие окончания поиска:

$$\left\|\Delta f\left(\overline{x^k}\right)\right\| = \sqrt{\sum_{i=1}^{n}\left(\frac{\partial f\left(\overline{x^k}\right)}{\partial x_i}\right)} \leq \varepsilon$$

2.2 Программная реализация

На Рисунке 2.1 представлен результат работы алгоритма градиентного спуска. Для достижения глобального минимума алгоритму понадобилось 13 итераций.

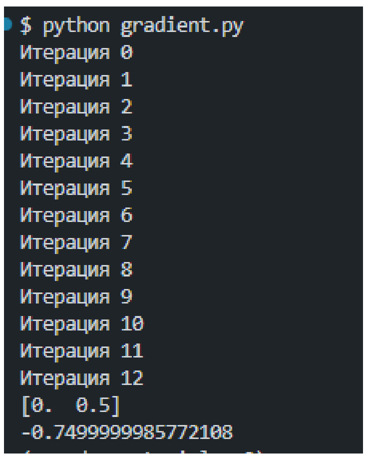

Рисунок 2.1 — Результат работы градиентного спуска

Реализация градиентного спуска представлена в Листинге Б.1.

3 МЕТОДЫ ВТОРОГО ПОРЯДКА

3.1 Теоретическая информация

Методы второго порядка используют производные второго порядка, что позволяют гарантировать сходимость к глобальному минимуму выпуклых функций.

Основным преимуществом методов второго порядка является их высокая сходимость - для квадратичных функций доказана сходимость за одну итерацию.

Координаты новой точки в данном методе вычисляются по Формуле 3.1.

$$\overline{x^{k+1}} = \overline{x^k} + \overline{p^k} \tag{3.1}$$

где k — номер итерации;

$p\left(\overline{x^k}\right) = -H^{-1}\left(\overline{x^k}\right)\Delta f\left(\overline{x^k}\right)$ — вектор направления спуска;

$H\left(\overline{x^k}\right)$ — матрица Гессе (матрица вторых производных).

Условие окончания поиска:

$$\left\|\Delta f\left(\overline{x^k}\right)\right\| = \sqrt{\sum_{i=1}^{n}\left(\frac{\partial f\left(\overline{x^k}\right)}{\partial x_i}\right)} \leq \varepsilon$$

3.2 Программная реализация

На Рисунке 3.1 представлен результат работы алгоритма градиентного спуска. Для достижения глобального минимума алгоритму понадобилась 1 итерация.

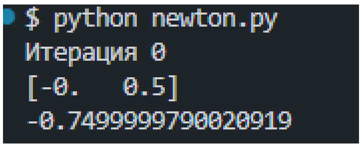

Рисунок 3.1 — Результат работа метода Ньютона

Реализация метода Ньютона представлена в Листинге В.1.

ЗАКЛЮЧЕНИЕ

После тестирования реализованных методов подтверждена теоретическая информация о них — метод нулевого порядка прост в реализации, однако имеет низкую скорость сходимости (15 итераций). Метод первого порядка (градиентный спуск с дроблением шага) уже улучшил скорость сходимости (13 итераций), а самым быстрым оказался метод Ньютона, для которого удалось подтвердить, что он сходится к глобальному минимуму квадратичной функции за одну итерацию.

Приложение А

Реализация метода Хука-Дживса

Листинг А.1 — Код реализации метода Хука-Дживса

```
import numpy as np

def f(arr: np.ndarray[np.float32]) -> float | np.ndarray[np.float32]:
    x = arr.T
    return 2 * x[0] ** 2 - 2 * x[0] * x[1] + 3 * x[1] ** 2 + x[0] - 3 * x[1]


def main():
    basis: np.ndarray[np.float32] = np.array([1., 1.])
    n = 2
    d = 10
    h = 0.2
    m = 2
    it = 0
    epsilon = 0.0001

    while True:
        new_x = basis.copy()
        while True:
            print(f'Итерация {it}')
            it += 1
            for i in range(n):
                basis = new_x.copy()
                new_x[i] += h
                if f(new_x) < f(basis):
                    continue
                new_x[i] -= 2 * h
                if f(new_x) < f(basis):
                    continue
                new_x[i] += h
            if (new_x == basis).all():
                h /= d
            else:
                break
        pattern = new_x + m * (new_x - basis)
        if f(pattern) < f(new_x):
            basis = pattern
        else:
            basis = new_x
        print(basis)
        print(f(basis))

        if h <= epsilon:
            print((lambda x: np.round(x, 3))(basis))
            print(f(basis))
            break



if __name__ == '__main__':
    main()
```

Приложение Б

Реализация градиентного спуска с дроблением шага

Листинг Б.1 — Код реализации градиентного спуска

```
import numpy as np


delta_x = 0.0000001


def f(arr: np.ndarray[np.float32]) -> float | np.ndarray[np.float32]:
    x = arr.T
    return 2 * x[0] ** 2 - 2 * x[0] * x[1] + 3 * x[1] ** 2 + x[0] - 3 * x[1]


def first_partial(x: np.ndarray[np.float32], i: int):
    u = x.copy()
    u[i] += delta_x
    u = f(u)

    return (u - f(x)) / delta_x


def gradient(x: np.ndarray[np.float32]) -> np.ndarray[np.float32]:
    result = np.array([first_partial(x, 0), first_partial(x, 1)])
    return result


def main():
    x = np.array([1., 1.])
    it = 0
    h = 0.2
    epsilon = 0.0001

    while np.linalg.norm(gradient(x)) > epsilon:
        while True:
            print(f'Итерация {it}')
            it += 1

            new_x = x - h * gradient(x)
            if f(new_x) < f(x):
                x = new_x.copy()
                break
            else:
                h /= 2

    print(np.round(x, 2))
    print(f(x))


if __name__ == '__main__':
    main()
```

Приложение В

Реализация метода Ньютона

Листинг В.1 — Код реализации метода Ньютона

```
import numpy as np

delta_x = 0.0001


def f(arr: np.ndarray[np.float32]) -> float | np.ndarray[np.float32]:
    x = arr.T
    return 2 * x[0] ** 2 - 2 * x[0] * x[1] + 3 * x[1] ** 2 + x[0] - 3 * x[1]


def first_partial(x: np.ndarray[np.float32], i: int):
    u = x.copy()
    u[i] += delta_x
    u = f(u)

    return (u - f(x)) / delta_x


def second_partial(x: np.ndarray[np.float32], i: int):
    u1 = x.copy()
    u1[i] += delta_x
    u1 = f(u1)

    u2 = f(x)

    u3 = x.copy()
    u3[i] -= delta_x
    u3 = f(u3)

    return (u1 - 2 * u2 + u3) / delta_x ** 2


def mixed_partial(x: np.ndarray[np.float32], i: int, j: int):
    u1 = f(x)

    u2 = x.copy()
    u2[i] -= delta_x
    u2 = f(u2)

    u3 = x.copy()
    u3[j] -= delta_x
    u3 = f(u3)

    u4 = x.copy()
    u4[i] -= delta_x
    u4[j] -= delta_x
    u4 = f(u4)

    return (u1 - u2 - u3 + u4) / delta_x ** 2


def hesse_matrix(x: np.ndarray[np.float32]) -> np.ndarray[np.float32]:
    matrix = []
    for i in range(len(x)):
        matrix.append([])
        for j in range(len(x)):
            if i == j:
                matrix[i].append(second_partial(x, i))
            else:
                matrix[i].append(mixed_partial(x, i, j))
    return np.array(matrix)


def check_positive(matrix: np.ndarray[np.float32]):
    minors = [] # A MINOOOOOOOOOOOOOOOOOR
```

Окончание Листинга В.1

```
    for i in range(len(matrix)):
        minors.append(np.linalg.det(matrix[:i + 1, :i + 1]))
    if all(x > 0 for x in minors):
        return True
    else:
        flag = True
        for i in range(len(minors)):
            if minors[i] * (-1) ** (i + 1) > 0:
                pass
            else:
                flag = False
                break
        if flag:
            return False
        else:
            raise Exception('Матрица Гессе не является положительно или
отрицательно определённой')


def calc_step(x: np.ndarray[np.float32]):
    result = np.square(np.linalg.norm(x)) / np.dot(hesse_matrix(x) @
gradient(x).T, gradient(x).T)
    return result


def gradient(x: np.ndarray[np.float32]) -> np.ndarray[np.float32]:
    result = np.array([first_partial(x, 0), first_partial(x, 1)])
    return result


def main():
    x = np.array([1., 1.])
    it = 0
    epsilon = 0.0001

    while np.linalg.norm(gradient(x)) > epsilon:
        print(f'Итерация {it}')
        it += 1

        hesse = hesse_matrix(x)
        if check_positive(hesse):
            x = x - np.linalg.inv(hesse) @ gradient(x).T
        else:
            x = x - calc_step(x) * gradient(x)

    print(np.round(x, 3))
    print(f(x))


if __name__ == '__main__':
    main()
```