МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра информационных технологий**

**ОТЧЁТ**

Работу выполнил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ М. В. Сидоренко

(подпись)

Направление подготовки 02.03.03 Математическое обеспечение и администрирование информационных систем

Направленность Методы оптимизации

Руководитель А.С. Чёрная

(подпись)

Краснодар

2024

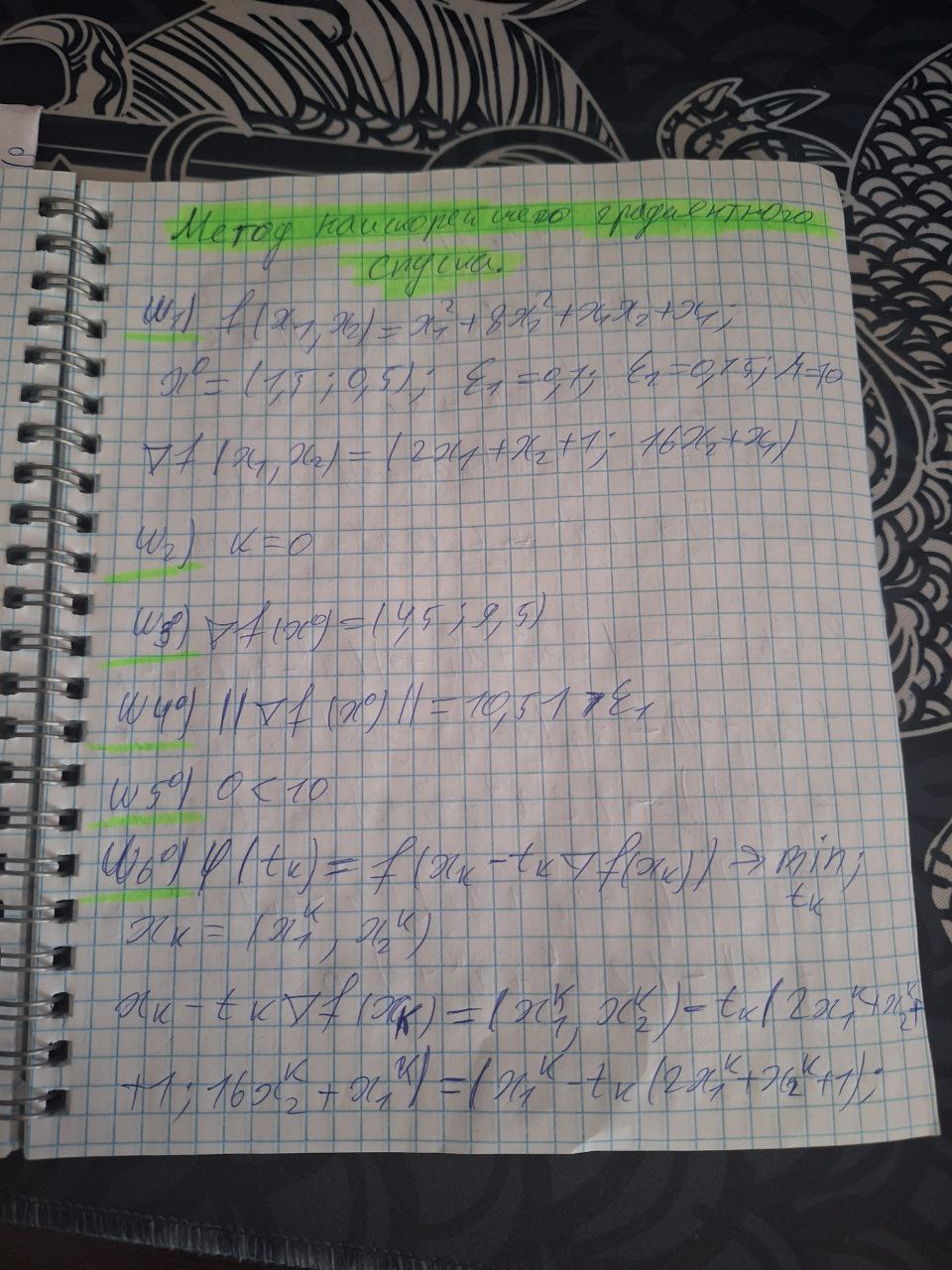
1 Постановка задачи

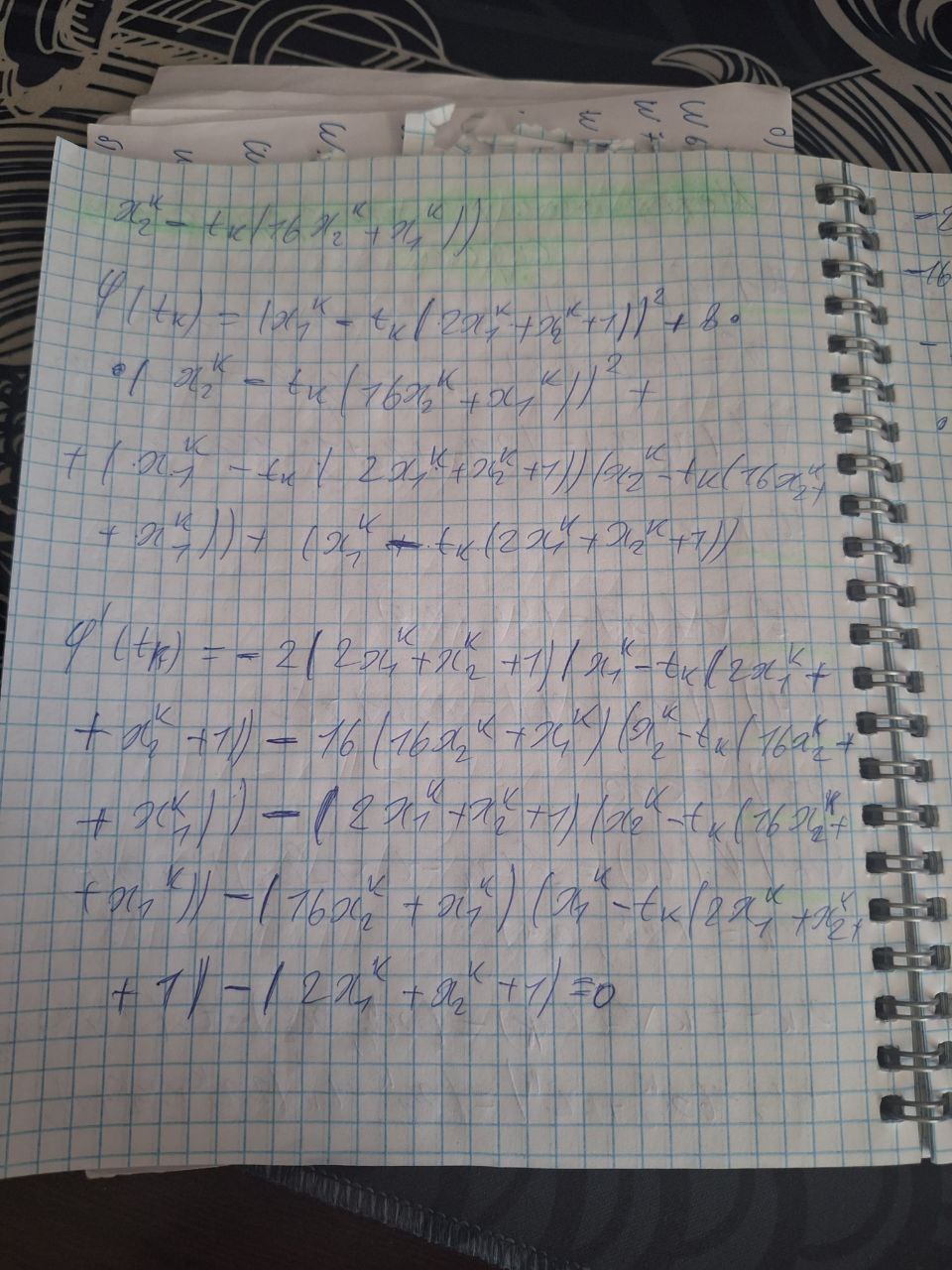
Требуется найти локальный минимум на множестве допустимых решений , т. е. такую точку , что

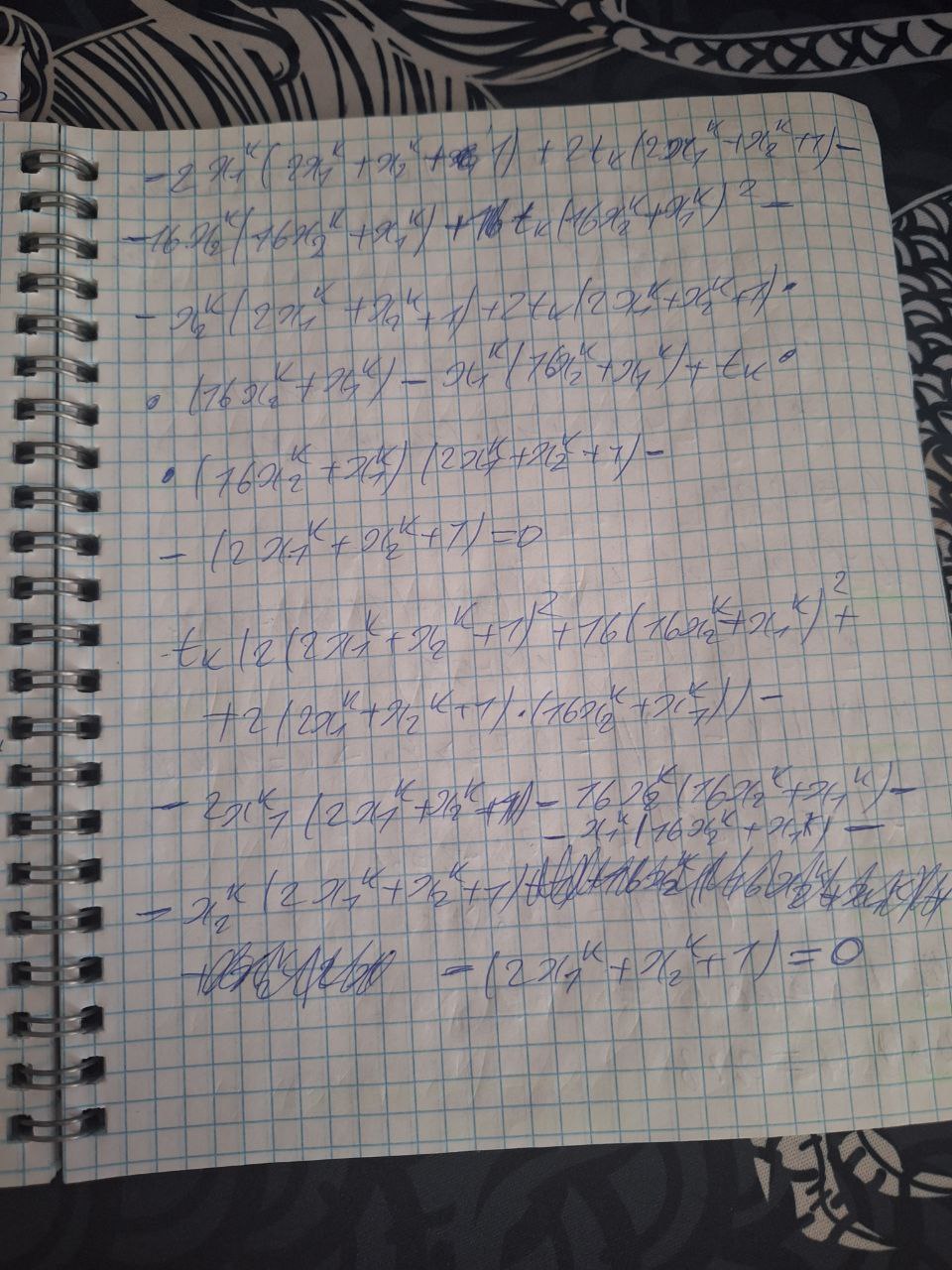
**Входные данные вариант 17:**

2 Метод наискорейшего градиентного спуска

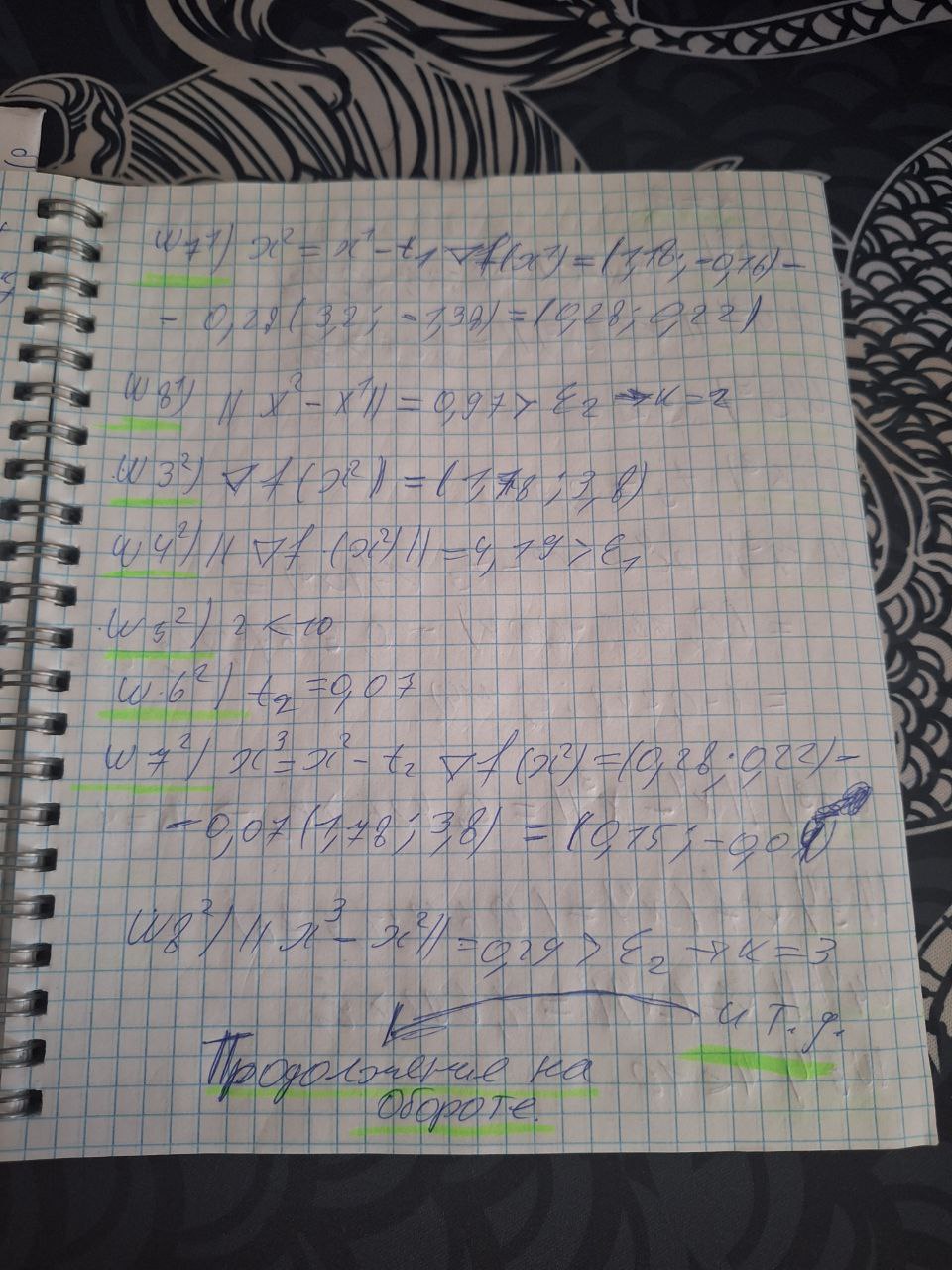
2.1 Пример работы алгоритма

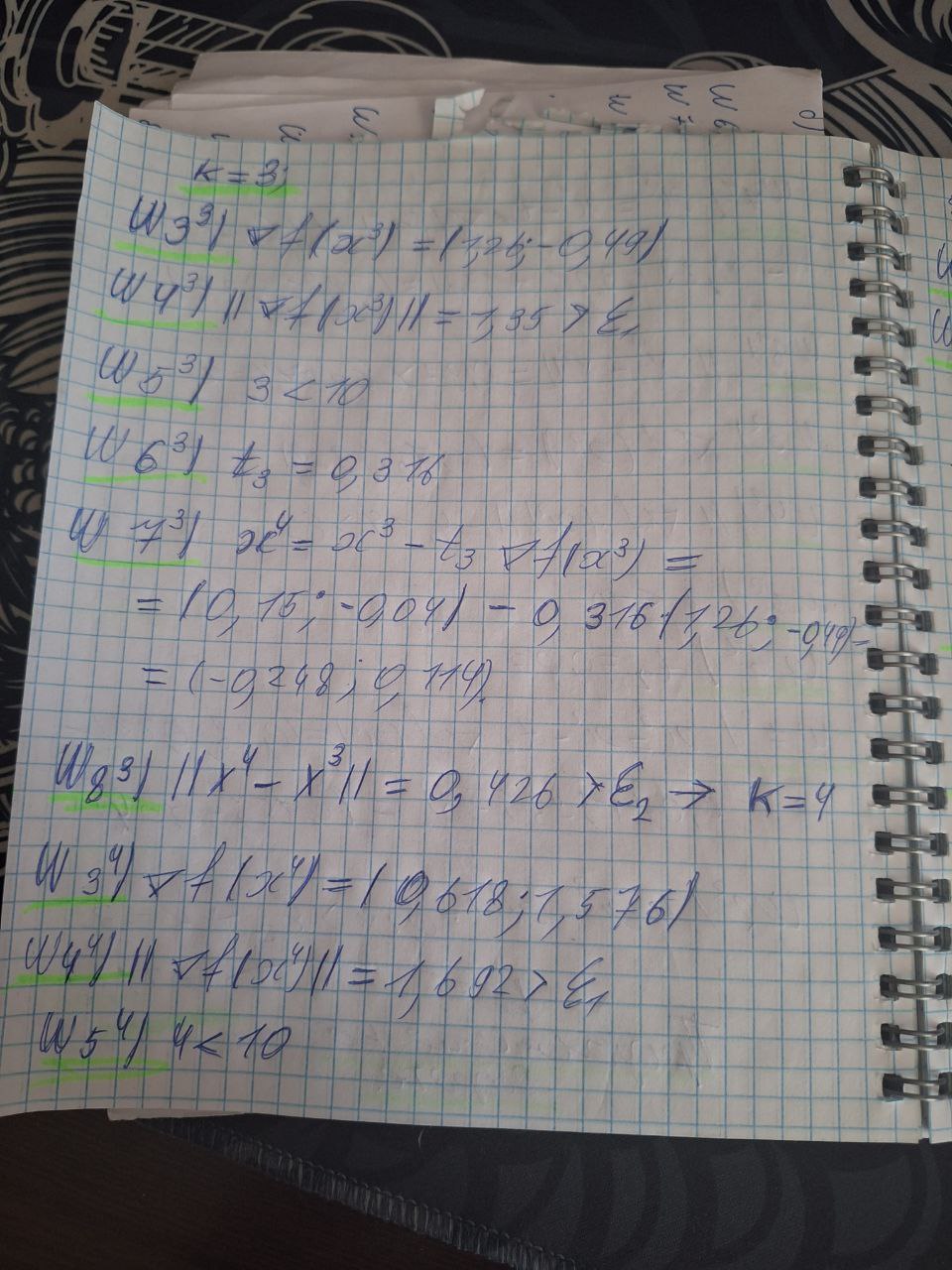


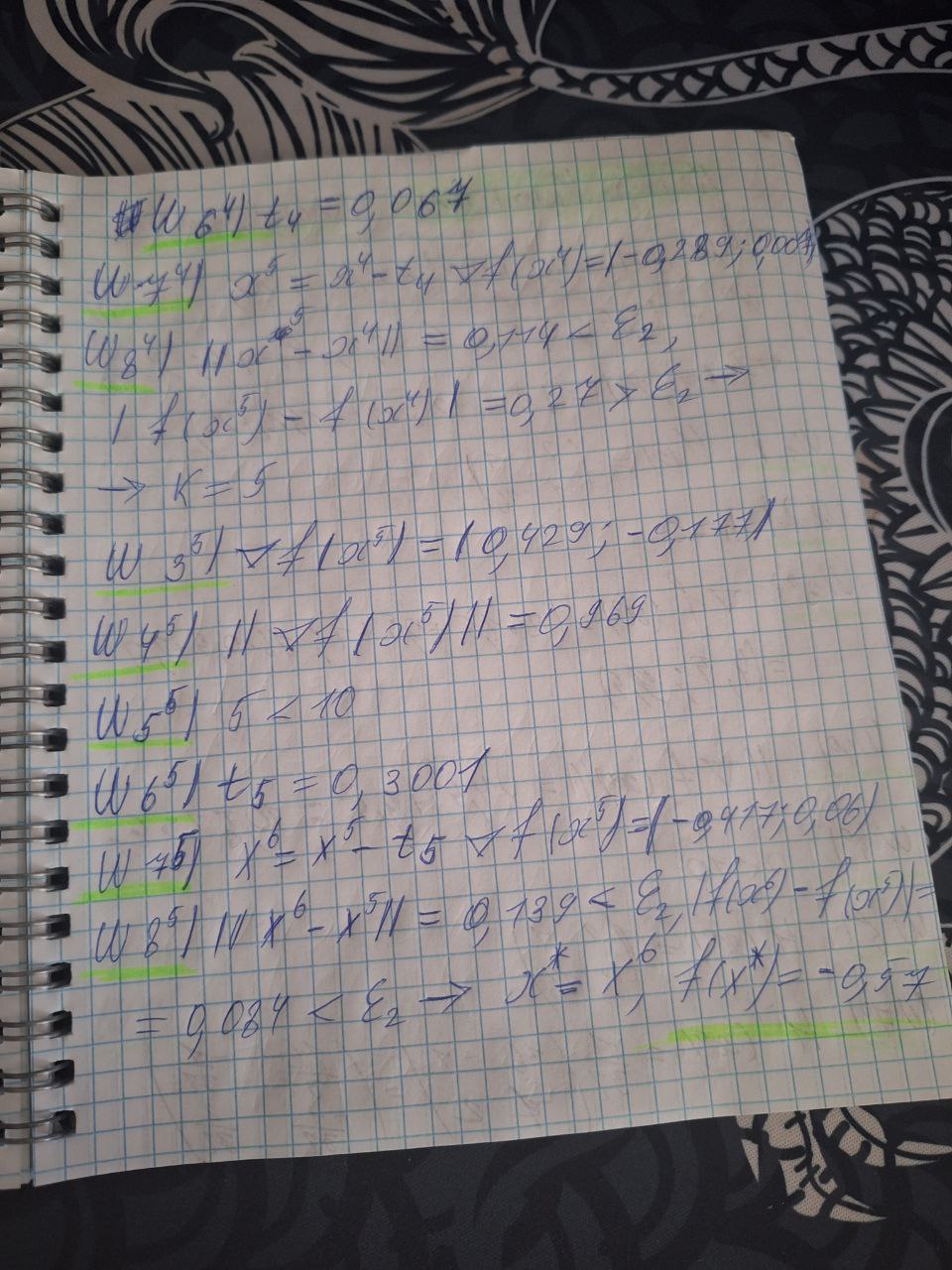












2.2 Код программы

import numpy as np

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

def create\_x\_range() -> list[list[*float*]]:

    """

    Функция, которая создаёт диапазон для функции

    Args: отсутствует

    Return: список списков с точками

    """

    x\_range = []

    step = 10 / 500

    for i in range(1000):

        x\_range.append([-5 + (step \* i), -5 + (step \* i)])

    return x\_range

def f(*x\_list*: list[list[*float*]]) -> list[*float*]:

    """

    Функция двух переменных

    Args: x - список значений x

    Return: список значений функции

    """

    result = []

    for x in x\_list:

        result.append([x[0]\*\*2 + 8\*(x[1]\*\*2) + x[0]\*x[1] + x[0]])

    return result

def grad\_f(*x*: list[*float*]) -> list[*float*]:

    """

    Функция, которая считает градиент функции f

    Args: x - аргумент функции

    Return: список значений

    """

    return [2\*x[0] + x[1] + 1, 16\*x[1] + x[0]]

def norm\_grad(*grad*: list[*float*]) -> *float*:

    """

    Функция, которая считает норму градиента

    Args: grad - градиент функции

    Return: значение нормы градиента функции

    """

    return np.sqrt(grad[0]\*\*2 + grad[1]\*\*2)

def calculate\_t(*x*: list[*float*]) -> *float*:

    """

    Функция, которая вычисляет значение t на какой-то итерации

    Args: x - аргумент

    Return: значение t

    """

    return (2\*x[0]\*(2\*x[0]+x[1]+1) + 16\*x[1]\*(16\*x[1]+x[0]) + x[1]\*(2\*x[0]+x[1]+1) + x[0]\*(16\*x[1]+x[0]) + 2\*x[0]+x[1]+1) / (2\*((2\*x[0]+x[1]+1)\*\*2) + 16\*((16\*x[1]+x[0])\*\*2) + 2\*(2\*x[0]+x[1]+1)\*(16\*x[1]+x[0]))

def calculate\_new\_x(*x\_old*: list[*float*], *t*: *float*) -> list[*float*]:

    """

    Функция, которая вычисляет новое значение х

    Args: x - значение старого x; t - значение t

    Return: новое значение x

    """

    return [x\_old[0] - t \* grad\_f(x\_old)[0], x\_old[1] - t \* grad\_f(x\_old)[1]]

def norm\_between\_x(*x\_old*: list[*float*], *x\_new*: list[*float*]) -> *float*:

    """

    Функция, которая вычисляет норму между новым и старым х

    Args: x\_old и x\_new - старый и новый х

    Return: значение нормы

    """

    return np.sqrt((x\_new[0] - x\_old[0])\*\*2 + (x\_new[1] - x\_old[1])\*\*2)

def mod\_between\_f(*x\_old*: list[*float*], *x\_new*: list[*float*]) -> *float*:

    """

    Функция, которая вычисляет модуль между разницей новой и старой f

    Args: x\_old и x\_new - старый и новый х

    Return: значение модуля

    """

    return abs(f([x\_new])[0][0] - f([x\_old])[0][0])

# Создать списки для сохранения значений x и f(x)

x\_history = []

f\_history = []

x\_0 = [1.5, 0.5]

epsilon\_1 = 0.1

epsilon\_2 = 0.15

m = 10

k = 0

flag = 0

while k < m:

    # Добавить текущее значение x и f(x) в историю

    x\_history.append(x\_0)

    f\_history.append(f([x\_0])[0][0])

    #шаг 3

    grad\_f\_x\_0 = grad\_f(x\_0)

    #шаг 4

    norm\_grad\_f\_x\_0 = norm\_grad(grad\_f\_x\_0)

    if norm\_grad\_f\_x\_0 < epsilon\_1:

        x\_min = x\_0

        f\_min = f([x\_min])

        break

    else:

        #шаг 5

        if k >= m:

            x\_min = x\_0

            f\_min = f([x\_min])

            break

        else:

            #шаг 6

            t = calculate\_t(x\_0)

            #шаг 7

            x\_new = calculate\_new\_x(x\_0, t)

            #шаг 8

            between\_x = norm\_between\_x(x\_0, x\_new)

            between\_f = mod\_between\_f(x\_0, x\_new)

            if between\_x < epsilon\_2 and between\_f < epsilon\_2:

                if flag == 1:

                    flag = 2

                    x\_min = x\_0

                    f\_min = f([x\_min])

                    break

                else:

                    flag = 1

                    x\_0 = x\_new

                    k += 1

            else:

                x\_0 = x\_new

                k += 1

print(f"Координаты точки минимума: {x\_min};\n f(x): {f\_min};\n k: {k}")

# Изменить часть для построения графика

x\_range = create\_x\_range()

x1\_values = np.array([x\_range[i][0] for i in range(len(x\_range))])

x2\_values = x1\_values.copy()

x, y = np.meshgrid(x1\_values, x2\_values)

z = np.array(f(x\_range))

fig = plt.figure()

ax = fig.add\_subplot(1, 1, 1, *projection*="3d")

ax.plot\_surface(x, y, z, *cmap*='viridis', *alpha*=0.6)

ax.scatter(x\_min[0], x\_min[1], f\_min, *color*="red")

# Добавить отображение шагов минимизации

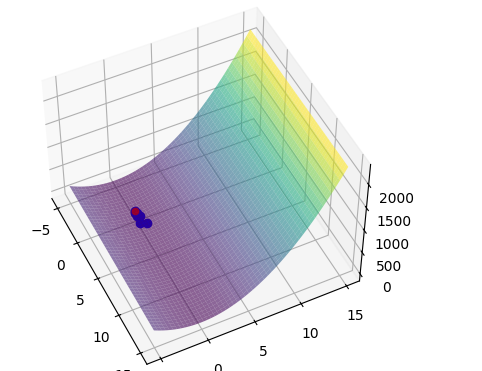
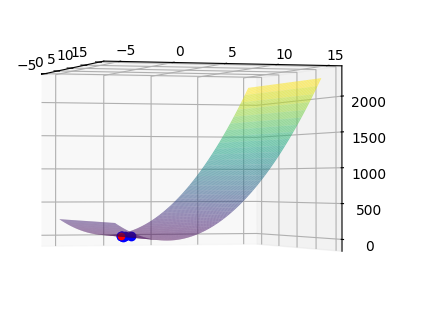
x\_history = np.array(x\_history)

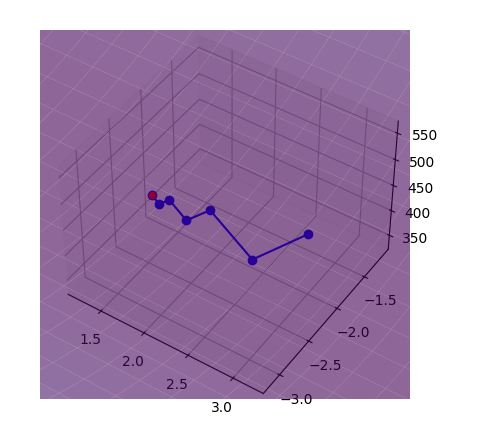
f\_history = np.array(f\_history)

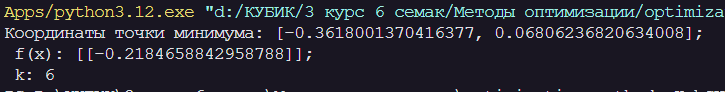
ax.plot(x\_history[:, 0], x\_history[:, 1], f\_history, *marker*='o', *color*='blue')

plt.show()

2.3 Вывод программы

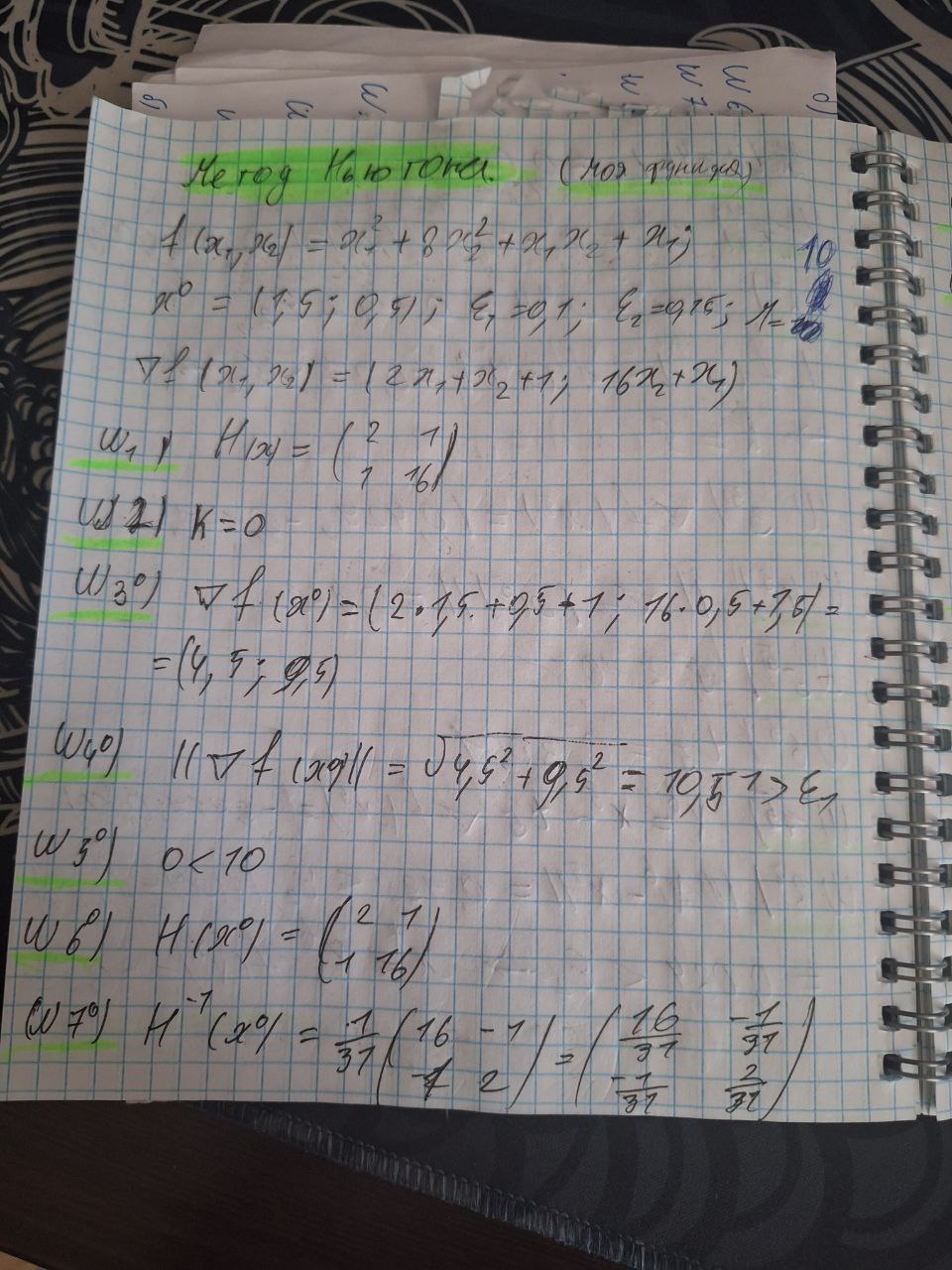
****

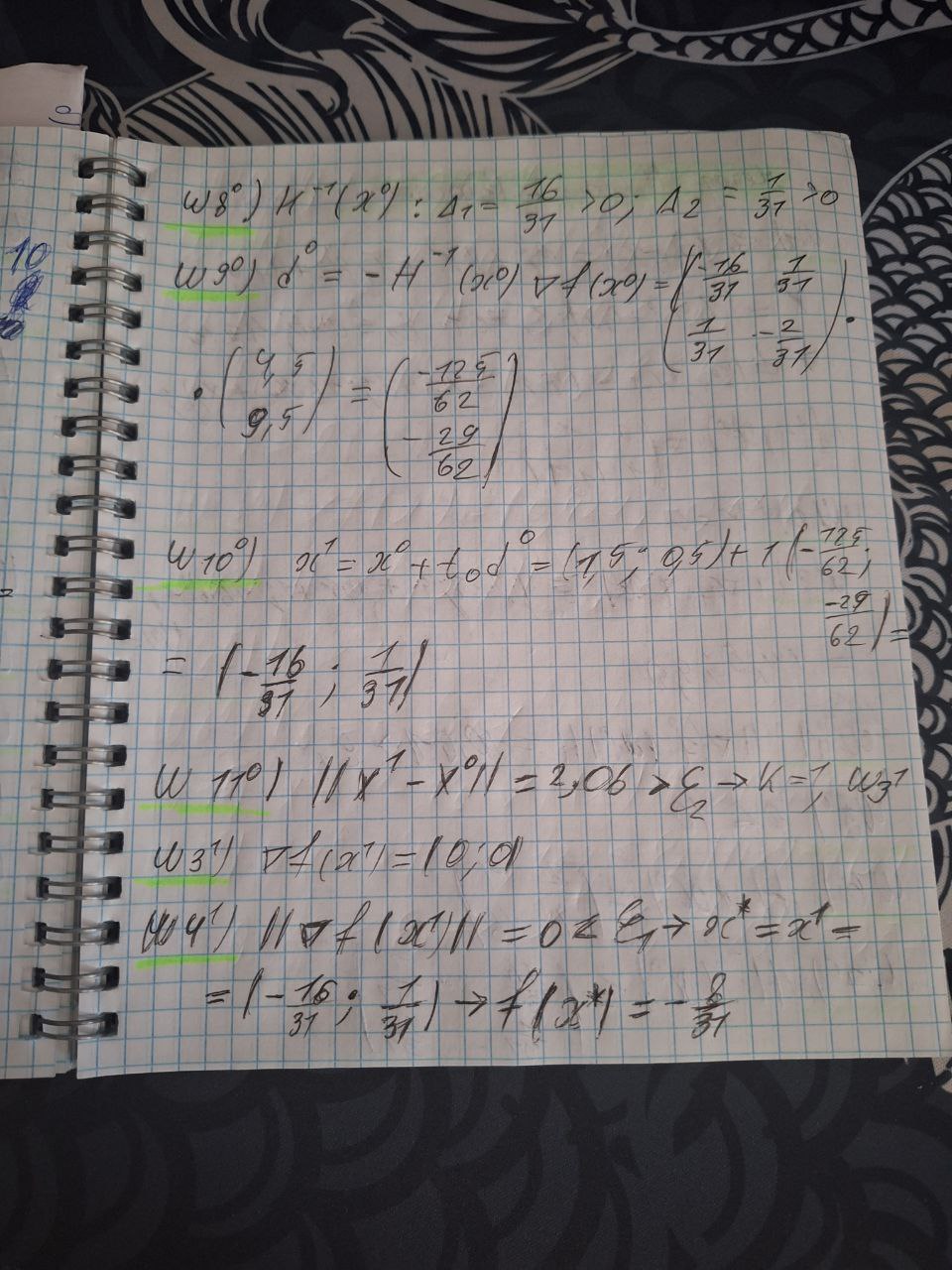
****

****

3 Метод Ньютона

3.1 Пример работы алгоритма





3.2 Код программы

import numpy as np

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

def create\_x\_range() -> list[list[*float*]]:

    """

    Функция, которая создаёт диапазон для функции

    Args: отсутствует

    Return: список списков с точками

    """

    x\_range = []

    step = 10 / 500

    for i in range(1000):

        x\_range.append([-5 + (step \* i), -5 + (step \* i)])

    return x\_range

def f(*x\_list*: list[list[*float*]]) -> list[*float*]:

    """

    Функция двух переменных

    Args: x - список значений x

    Return: список значений функции

    """

    result = []

    for x in x\_list:

        result.append([x[0]\*\*2 + 8\*(x[1]\*\*2) + x[0]\*x[1] + x[0]])

    return result

def grad\_f(*x*: list[*float*]) -> list[*float*]:

    """

    Функция, которая считает градиент функции f

    Args: x - аргумент функции

    Return: список значений

    """

    return [2\*x[0] + x[1] + 1, 16\*x[1] + x[0]]

def norm\_grad(*grad*: list[*float*]) -> *float*:

    """

    Функция, которая считает норму градиента

    Args: grad - градиент функции

    Return: значение нормы градиента функции

    """

    return np.sqrt(grad[0]\*\*2 + grad[1]\*\*2)

def reverse\_mat(*h*: list[list[*float*]]) -> list[list[*float*]]:

    """

    Функция, которая вычисляет обратную матрицу

    Args: h - матрица Гиссе

    Return: обратная матрица

    """

    return np.linalg.inv(h)

def calculate\_deltas(*h*: list[list[*float*]]) -> list[*float*]:

    """

    Функция, которая вычисляет 2 дельты матрицы

    Args: h - матрица Гиссе

    Return: список из двух дельт

    """

    return [h[0][0], np.linalg.det(h)]

def calculate\_d(*x*: list[*float*], *h*: list[list[*float*]]) -> list[*float*]:

    """

    Функция, которая вычисляет вектор d

    Args: x - список значений, h - матрица Гиссе (уже обратная)

    Return: список d

    """

    return np.dot(-h, grad\_f(x))

def calculate\_t(*x*: list[*float*]) -> *float*:

    """

    Функция, которая вычисляет значение t на какой-то итерации

    Args: x - аргумент

    Return: значение t

    """

    return (2\*x[0]\*(2\*x[0]+x[1]+1) + 16\*x[1]\*(16\*x[1]+x[0]) + x[1]\*(2\*x[0]+x[1]+1) + x[0]\*(16\*x[1]+x[0]) + 2\*x[0]+x[1]+1) / (2\*((2\*x[0]+x[1]+1)\*\*2) + 16\*((16\*x[1]+x[0])\*\*2) + 2\*(2\*x[0]+x[1]+1)\*(16\*x[1]+x[0]))

def calculate\_x\_new(*x\_old*: list[*float*], *t*: *float*, *d*: list[*float*]) -> list[*float*]:

    """

    Функция, которая вычисляет новое значение вектора x

    Args: x\_old - предыдущее значение вектора x; t - коэффициент t; d - коэффициент d

    Return: новый вектор x

    """

    return np.array(x\_old) + t \* np.array(d)

def norm\_between\_x(*x\_old*: list[*float*], *x\_new*: list[*float*]) -> *float*:

    """

    Функция, которая вычисляет норму между новым и старым х

    Args: x\_old и x\_new - старый и новый х

    Return: значение нормы

    """

    return np.sqrt((x\_new[0] - x\_old[0])\*\*2 + (x\_new[1] - x\_old[1])\*\*2)

def mod\_between\_f(*x\_old*: list[*float*], *x\_new*: list[*float*]) -> *float*:

    """

    Функция, которая вычисляет модуль между разницей новой и старой f

    Args: x\_old и x\_new - старый и новый х

    Return: значение модуля

    """

    return abs(f(x\_new)[0][0] - f(x\_old)[0][0])

# Создать списки для сохранения значений x и f(x)

x\_history = []

f\_history = []

x\_0 = [1.5, 0.5]

epsilon\_1 = 0.1

epsilon\_2 = 0.15

m = 10

k = 0

h = [[2, 1], [1, 16]]

flag = 0

while k < m:

    # Добавить текущее значение x и f(x) в историю

    x\_history.append(x\_0)

    f\_history.append(f([x\_0])[0][0])

    #шаг 3

    grad\_f\_x\_0 = grad\_f(x\_0)

    #шаг 4

    norm\_grad\_f\_x\_0 = norm\_grad(grad\_f\_x\_0)

    if norm\_grad\_f\_x\_0 < epsilon\_1:

        x\_min = x\_0

        f\_min = f([x\_min])

        break

    else:

        #шаг 5

        if k >= m:

            x\_min = x\_0

            f\_min = f(x\_min)

            break

        else:

            #шаг 6-7

            h\_reverse = reverse\_mat(h)

            #шаг 8

            deltas\_h\_reverse = calculate\_deltas(h\_reverse)

            if deltas\_h\_reverse[0] > 0 and deltas\_h\_reverse[1] > 0:

                #шаг 9

                d = calculate\_d(x\_0, h\_reverse)

                #шаг 10

                x\_new = calculate\_x\_new(x\_0, 1, d)

                #шаг 11

                between\_x = norm\_between\_x(x\_0, x\_new)

                between\_f = mod\_between\_f([x\_0], [x\_new])

                if between\_x < epsilon\_2 and between\_f < epsilon\_2:

                    if flag == 1:

                        flag = 2

                        x\_min = x\_new

                        f\_min = f(x\_min)

                        break

                    else:

                        flag = 1

                        x\_0 = x\_new

                        k += 1

                else:

                    x\_0 = x\_new

                    k += 1

            else:

                #шаг 8 б

                d = -grad\_f\_x\_0

                t = calculate\_t(x\_0)

                #шаг 10

                x\_new = calculate\_x\_new(x\_0, t, d)

                #шаг 11

                between\_x = norm\_between\_x(x\_0, x\_new)

                between\_f = mod\_between\_f([x\_0], [x\_new])

                if between\_x < epsilon\_2 and between\_f < epsilon\_2:

                    if flag == 1:

                        flag = 2

                        x\_min = x\_new

                        f\_min = f(x\_min)

                        break

                    else:

                        flag = 1

                        x\_0 = x\_new

                        k += 1

                else:

                    x\_0 = x\_new

                    k += 1

print(f"Координаты точки минимума: {x\_min};\n f(x): {f\_min};\n k: {k}")

# Изменить часть для построения графика

x\_range = create\_x\_range()

x1\_values = np.array([x\_range[i][0] for i in range(len(x\_range))])

x2\_values = x1\_values.copy()

x, y = np.meshgrid(x1\_values, x2\_values)

z = np.array(f(x\_range))

fig = plt.figure()

ax = fig.add\_subplot(1, 1, 1, *projection*="3d")

ax.plot\_surface(x, y, z, *cmap*='viridis', *alpha*=0.6)

ax.scatter(x\_min[0], x\_min[1], f\_min, *color*="red")

# Добавить отображение шагов минимизации

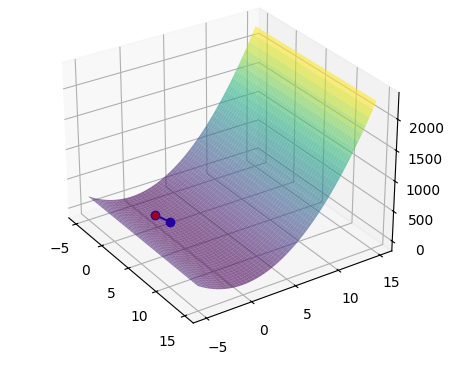
x\_history = np.array(x\_history)

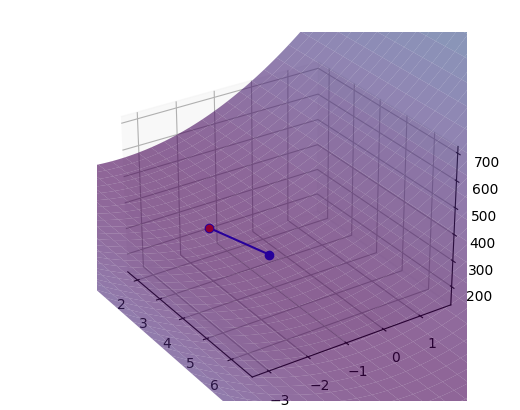
f\_history = np.array(f\_history)

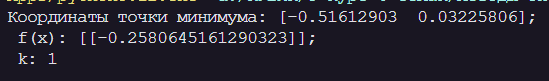
ax.plot(x\_history[:, 0], x\_history[:, 1], f\_history, *marker*='o', *color*='blue')

plt.show()

3.3 Вывод программы

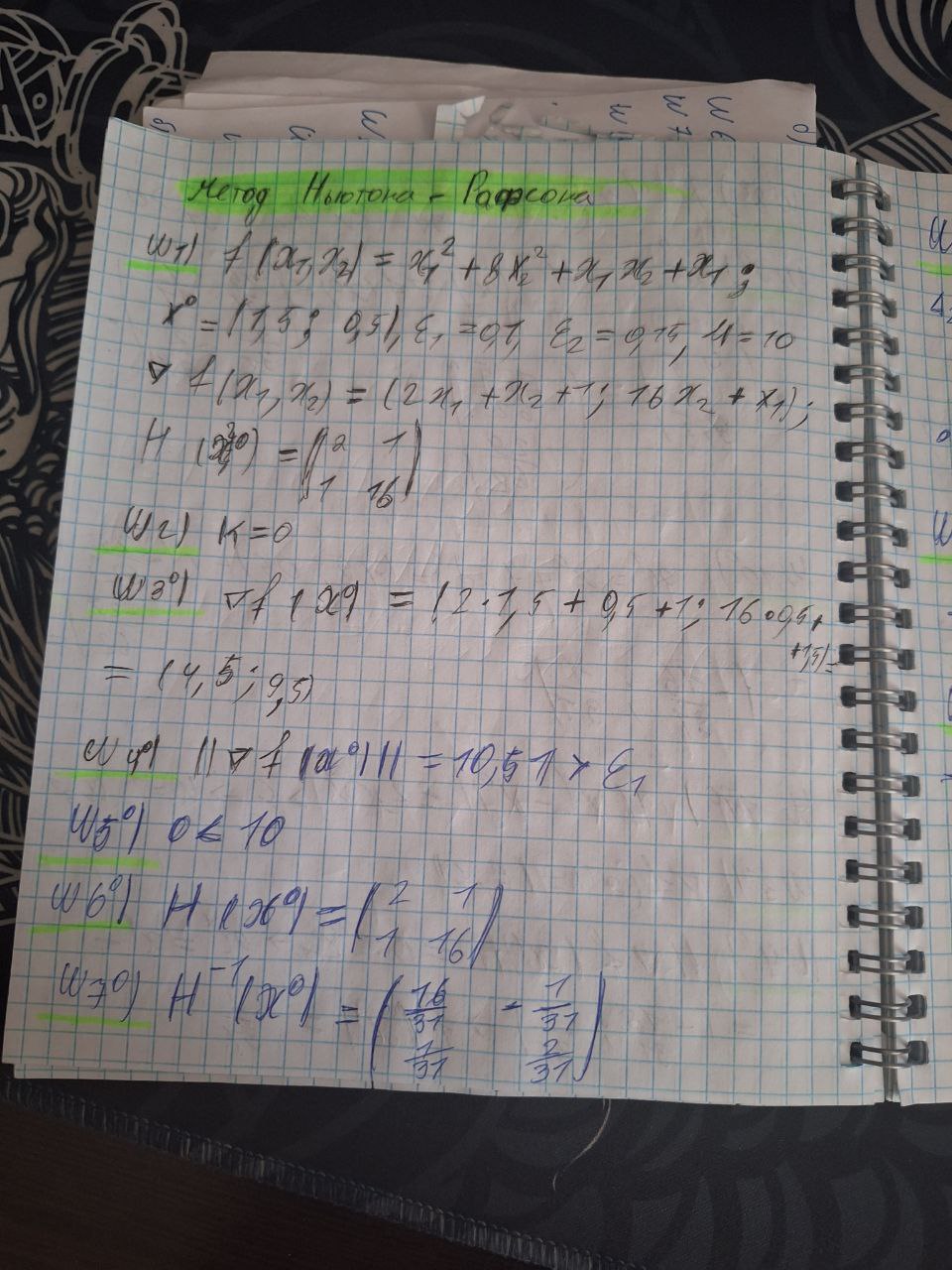
****

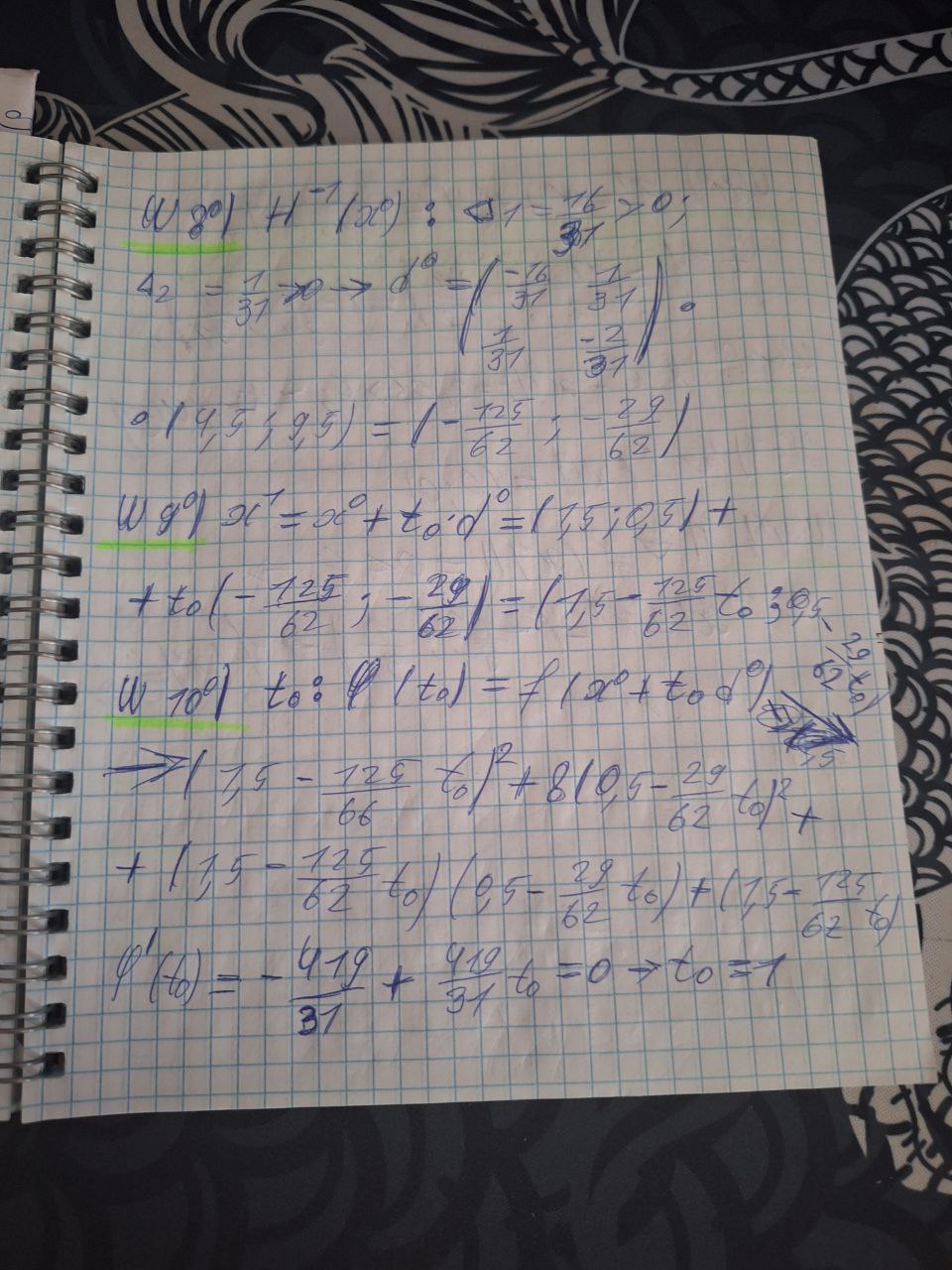
****

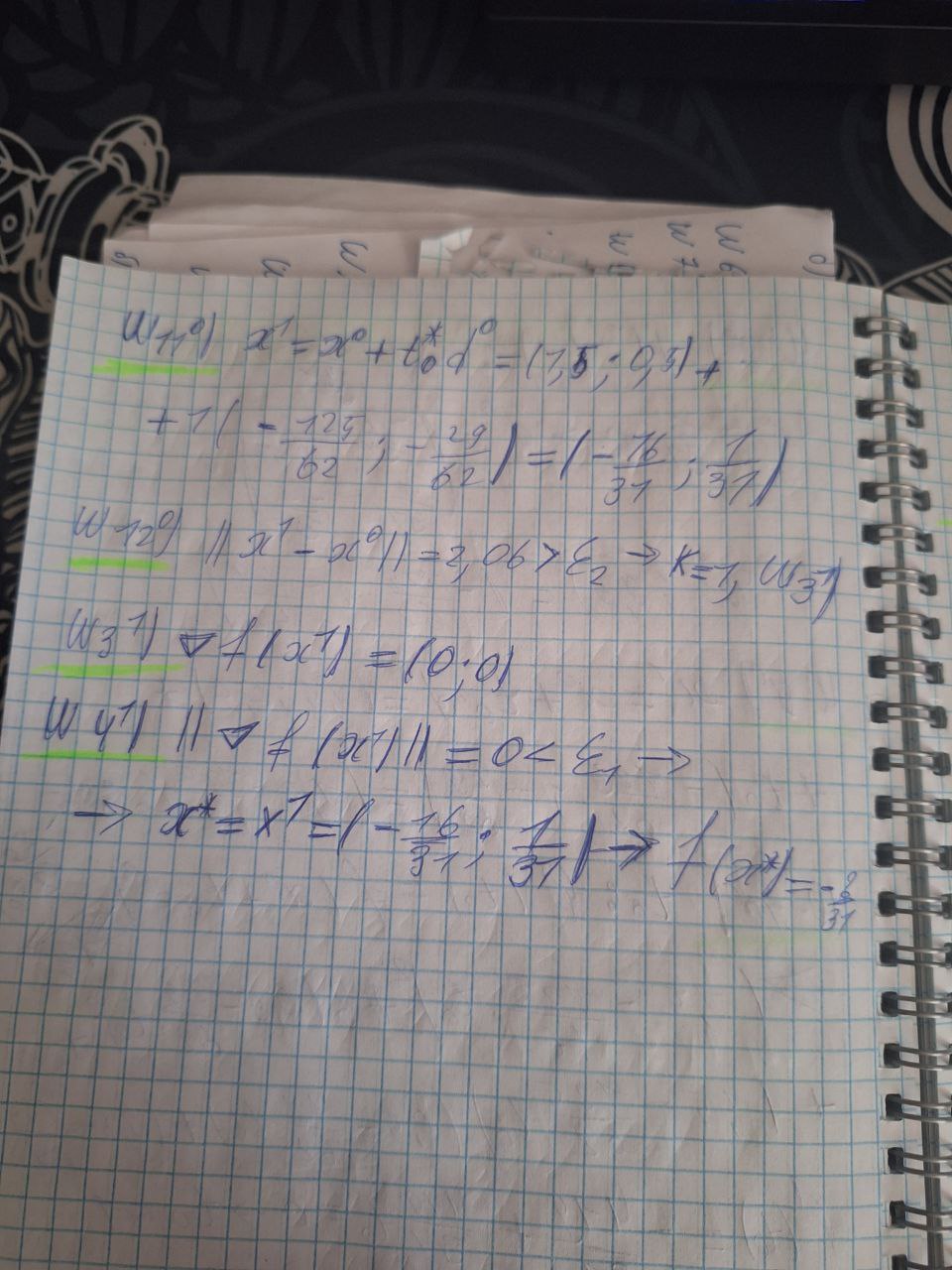
****

4 Метод Ньютона-Рафсона

4.1 Пример работы алгоритма







4.2 Код программы

import math as mt

import numpy as np

import sympy as sp

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

def create\_x\_range() -> list[list[*float*]]:

    """

    Функция, которая создаёт диапазон для функции

    Args: отсутствует

    Return: список списков с точками

    """

    x\_range = []

    step = 10 / 500

    for i in range(1000):

        x\_range.append([-5 + (step \* i), -5 + (step \* i)])

    return x\_range

def f(*x\_list*: list[list[*float*]]) -> list[*float*]:

    """

    Функция двух переменных

    Args: x - список значений x

    Return: список значений функции

    """

    result = []

    for x in x\_list:

        result.append([x[0]\*\*2 + 8\*(x[1]\*\*2) + x[0]\*x[1] + x[0]])

    return result

def grad\_f(*x*: list[*float*]) -> list[*float*]:

    """

    Функция, которая считает градиент функции f

    Args: x - аргумент функции

    Return: список значений

    """

    return [2\*x[0] + x[1] + 1, 16\*x[1] + x[0]]

def norm\_grad(*grad*: list[*float*]) -> *float*:

    """

    Функция, которая считает норму градиента

    Args: grad - градиент функции

    Return: значение нормы градиента функции

    """

    return mt.sqrt(grad[0]\*\*2 + grad[1]\*\*2)

def reverse\_mat(*h*: list[list[*float*]]) -> list[list[*float*]]:

    """

    Функция, которая вычисляет обратную матрицу

    Args: h - матрица Гиссе

    Return: обратная матрица

    """

    return np.linalg.inv(h)

def calculate\_deltas(*h*: list[list[*float*]]) -> list[*float*]:

    """

    Функция, которая вычисляет 2 дельты матрицы

    Args: h - матрица Гиссе

    Return: список из двух дельт

    """

    return [h[0][0], np.linalg.det(h)]

def calculate\_d(*x*: list[*float*], *h\_reverse*: list[list[*float*]], *f*: *int*) -> list[*float*]:

    """

    Функция, которая вычисляет вектор d

    Args: x - список значений, h\_reverse - матрица Гиссе (уже обратная), f - флаг для выбора варианта вычисления

    Return: список d

    """

    if f == 1:

        return -(np.dot(h\_reverse, grad\_f(x)))

    elif f == 2:

        return [-x for x in grad\_f(x)]

    else:

        print("Выбран неверный флаг! Можно указать 1 или 2")

def calculate\_t(*x\_old*: list[*float*], *d*: list[*float*]) -> *float*:

    """

    Функция, которая вычисляет значение t на какой-то итерации

    Args: x\_old - список значений; d - коэффициент d

    Return: значение t

    """

    t = sp.symbols("t")  #объявляем символьную переменную

    expression\_x\_new = [x\_old[0] + t \* d[0], x\_old[1] + t \* d[1]]

    after\_f =  (expression\_x\_new[0]\*\*2) + (8 \* (expression\_x\_new[1]\*\*2)) + (expression\_x\_new[0] \* expression\_x\_new[1]) + expression\_x\_new[0]

    after\_f\_derivative = sp.diff(after\_f, t)  #берём производную по t

    solution\_t = sp.solve(after\_f\_derivative, t)  #решение уравнения производной = 0 для t

    return solution\_t

def calculate\_x\_new(*x\_old*: list[*float*], *t*: *float*, *d*: list[*float*]) -> list[*float*]:

    """

    Функция, которая вычисляет новое значение вектора x

    Args: x\_old - предыдущее значение вектора x; t - коэффициент t; d - коэффициент d

    Return: новый вектор x

    """

    return np.array(x\_old) + t \* np.array(d)

def norm\_between\_x(*x\_old*: list[*float*], *x\_new*: list[*float*]) -> *float*:

    """

    Функция, которая вычисляет норму между новым и старым х

    Args: x\_old и x\_new - старый и новый х

    Return: значение нормы

    """

    return mt.sqrt((x\_new[0] - x\_old[0])\*\*2 + (x\_new[1] - x\_old[1])\*\*2)

def mod\_between\_f(*x\_old*: list[*float*], *x\_new*: list[*float*]) -> *float*:

    """

    Функция, которая вычисляет модуль между разницей новой и старой f

    Args: x\_old и x\_new - старый и новый х

    Return: значение модуля

    """

    return abs(f(x\_new)[0][0] - f(x\_old)[0][0])

# Создать списки для сохранения значений x и f(x)

x\_history = []

f\_history = []

x\_0 = [1.5, 0.5]

epsilon\_1 = 0.1

epsilon\_2 = 0.15

m = 10

k = 0

h = [[2, 1], [1, 16]]

flag = 0

while k < m:

    # Добавить текущее значение x и f(x) в историю

    x\_history.append(x\_0)

    f\_history.append(f([x\_0])[0][0])

    #шаг 3

    grad\_x\_0 = grad\_f(x\_0)

    #шаг 4

    norm\_grad\_x\_0 = norm\_grad(grad\_x\_0)

    if norm\_grad\_x\_0 < epsilon\_1:

        x\_min = x\_0

        f\_min = f([x\_min])

        break

    else:

        #шаг 5

        if k >= m:

            x\_min = x\_0

            f\_min = f([x\_min])

            break

        else:

            #шаг 6-7

            h\_reverse = reverse\_mat(h)

            #шаг 8

            deltas = calculate\_deltas(h\_reverse)

            if deltas[0] > 0 and deltas[1] > 0:

                d = calculate\_d(x\_0, h\_reverse, 1)

            else:

                d = calculate\_d(x\_0, h\_reverse, 2)

            #шаг 9-10

            t = calculate\_t(x\_0, d)

            #шаг 11

            x\_new = calculate\_x\_new(x\_0, t, d)

            #шаг 12

            between\_x = norm\_between\_x(x\_0, x\_new)

            between\_f = mod\_between\_f([x\_0], [x\_new])

            if between\_x < epsilon\_2 and between\_f < epsilon\_2:

                if flag == 1:

                    flag = 2

                    x\_min = x\_new

                    f\_min = f([x\_min])

                    break

                else:

                    flag = 1

                    x\_0 = x\_new

                    k += 1

            else:

                x\_0 = x\_new

                k += 1

print(f"Координаты точки минимума: {x\_min};\n f(x): {f\_min};\n k: {k}")

# Изменить часть для построения графика

x\_range = create\_x\_range()

x1\_values = np.array([x\_range[i][0] for i in range(len(x\_range))])

x2\_values = x1\_values.copy()

x, y = np.meshgrid(x1\_values, x2\_values)

z = np.array(f(x\_range))

fig = plt.figure()

ax = fig.add\_subplot(1, 1, 1, *projection*="3d")

ax.plot\_surface(x, y, z, *cmap*='viridis', *alpha*=0.6)

ax.scatter(x\_min[0], x\_min[1], f\_min, *color*="red")

# Добавить отображение шагов минимизации

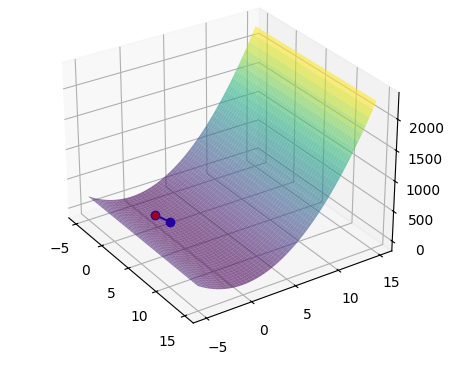
x\_history = np.array(x\_history)

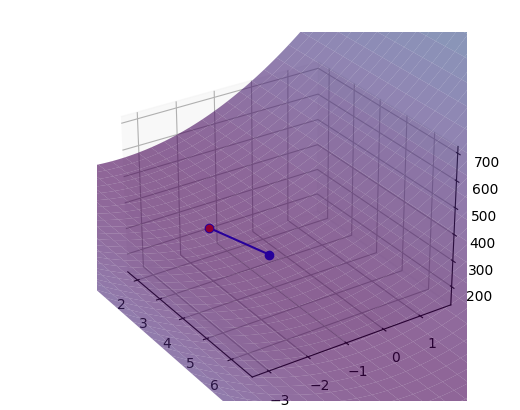
f\_history = np.array(f\_history)

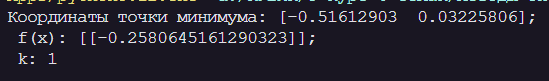
ax.plot(x\_history[:, 0], x\_history[:, 1], f\_history, *marker*='o', *color*='blue')

plt.show()

**4.3 Вывод программы**

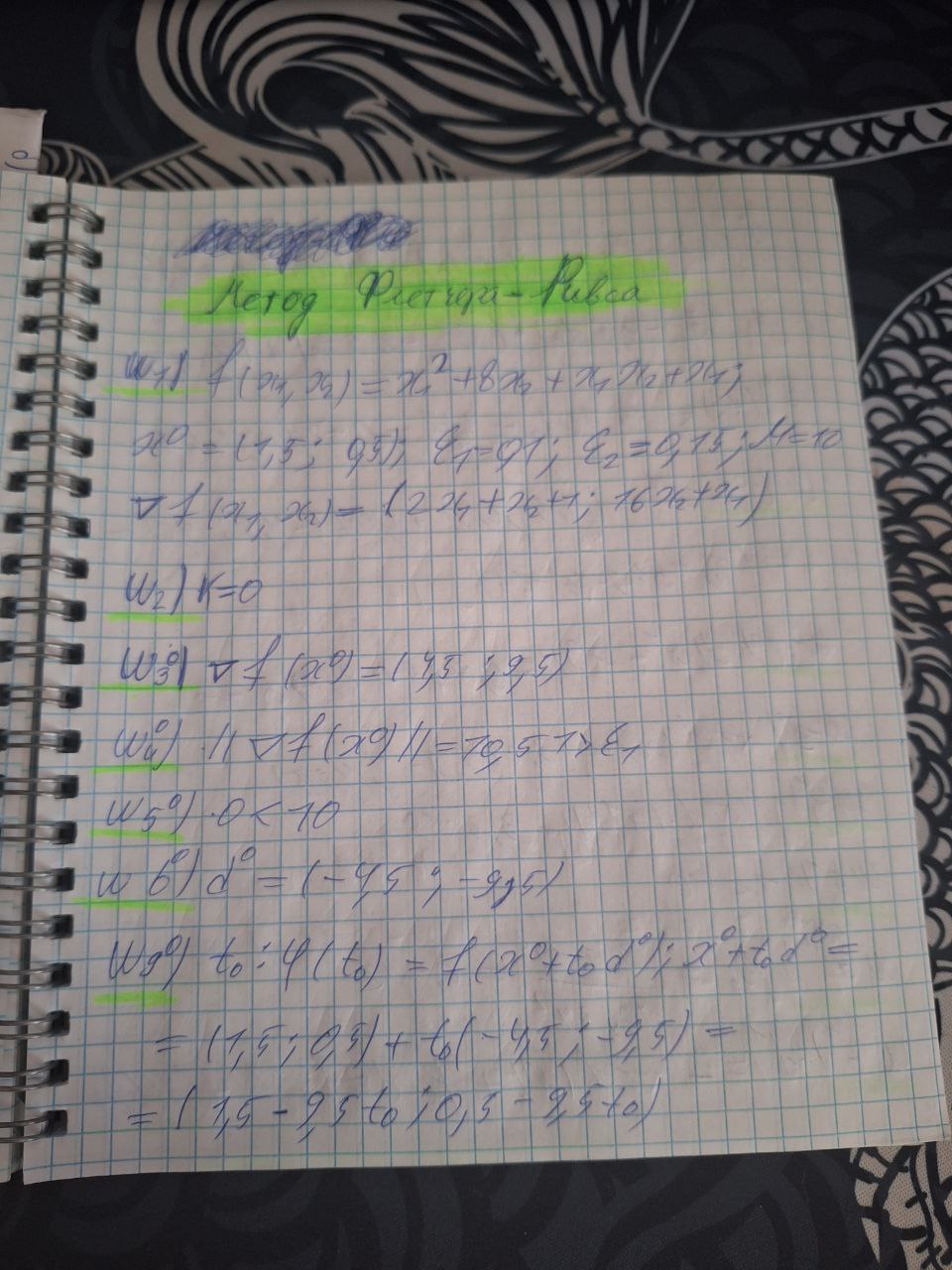
****

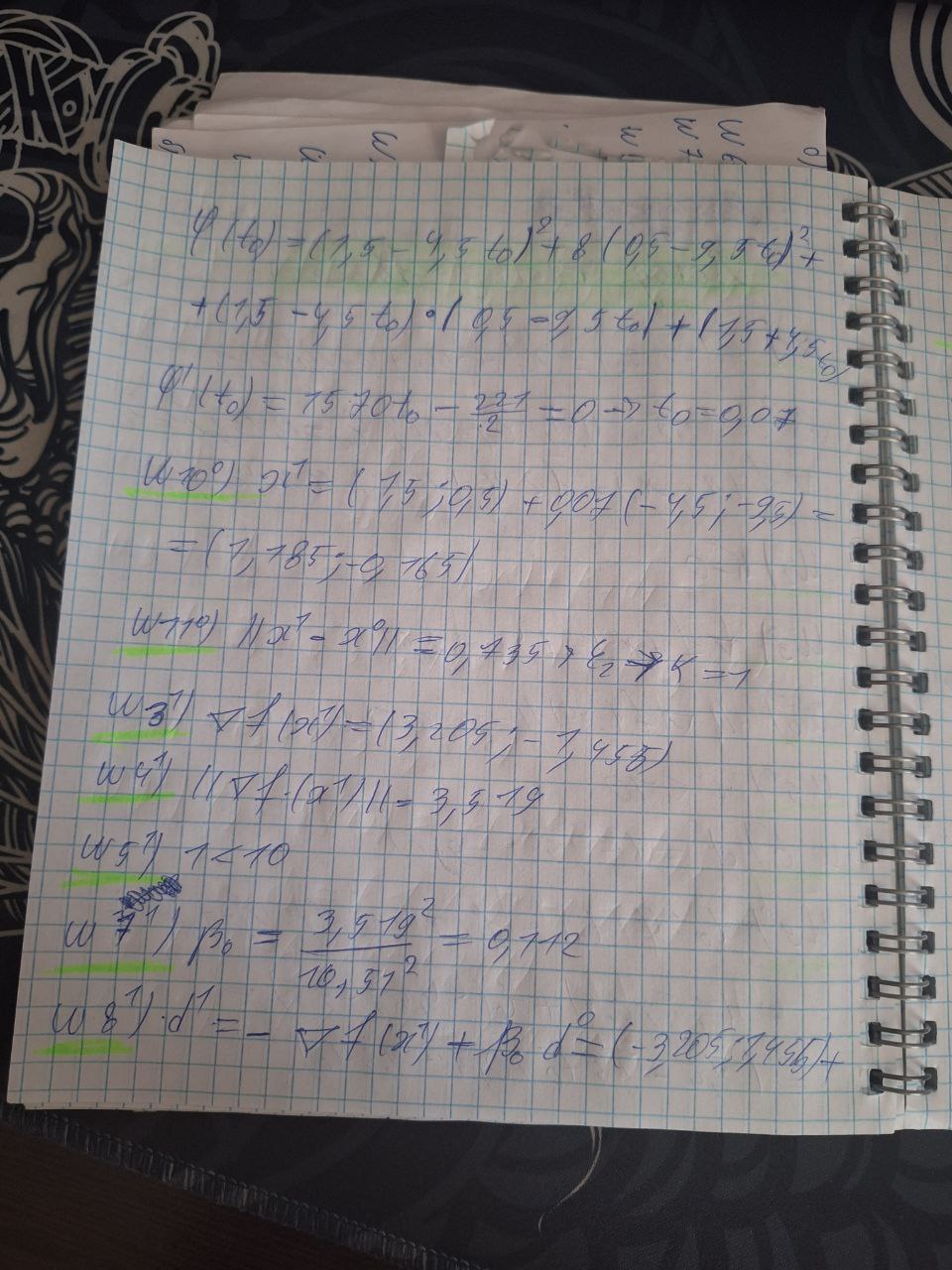
****

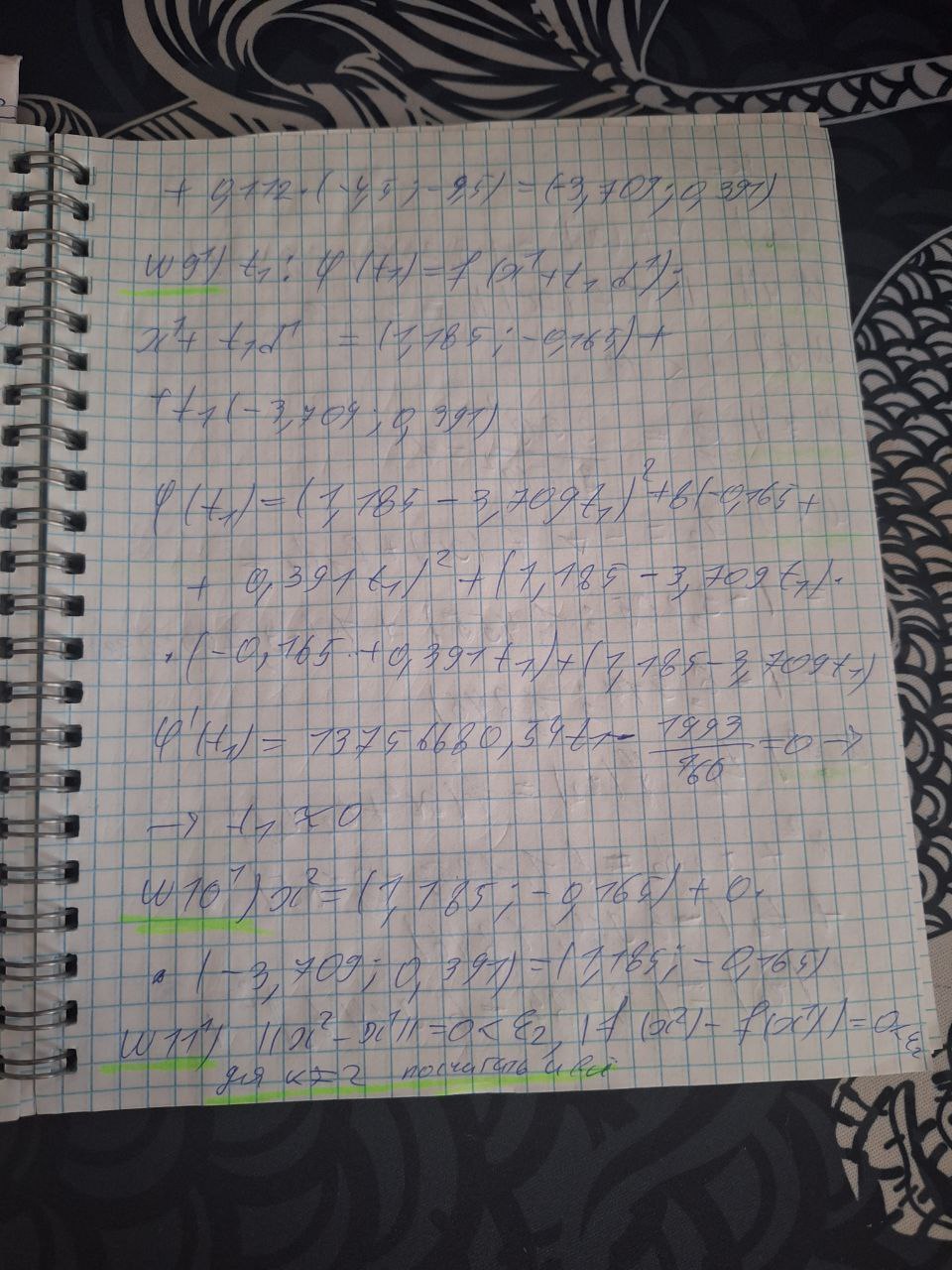
****

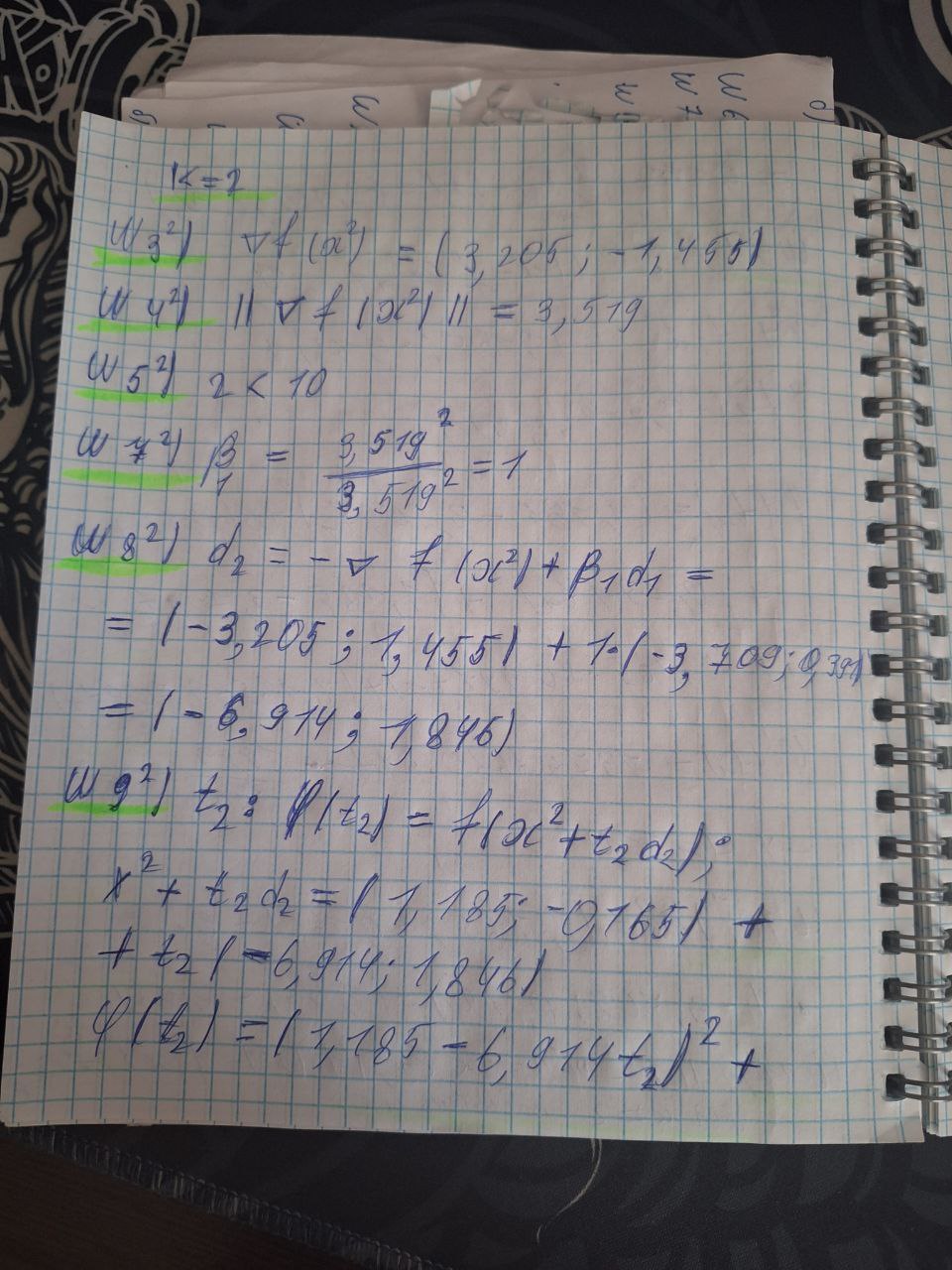
**5 Метод Флетчера-Ривса**

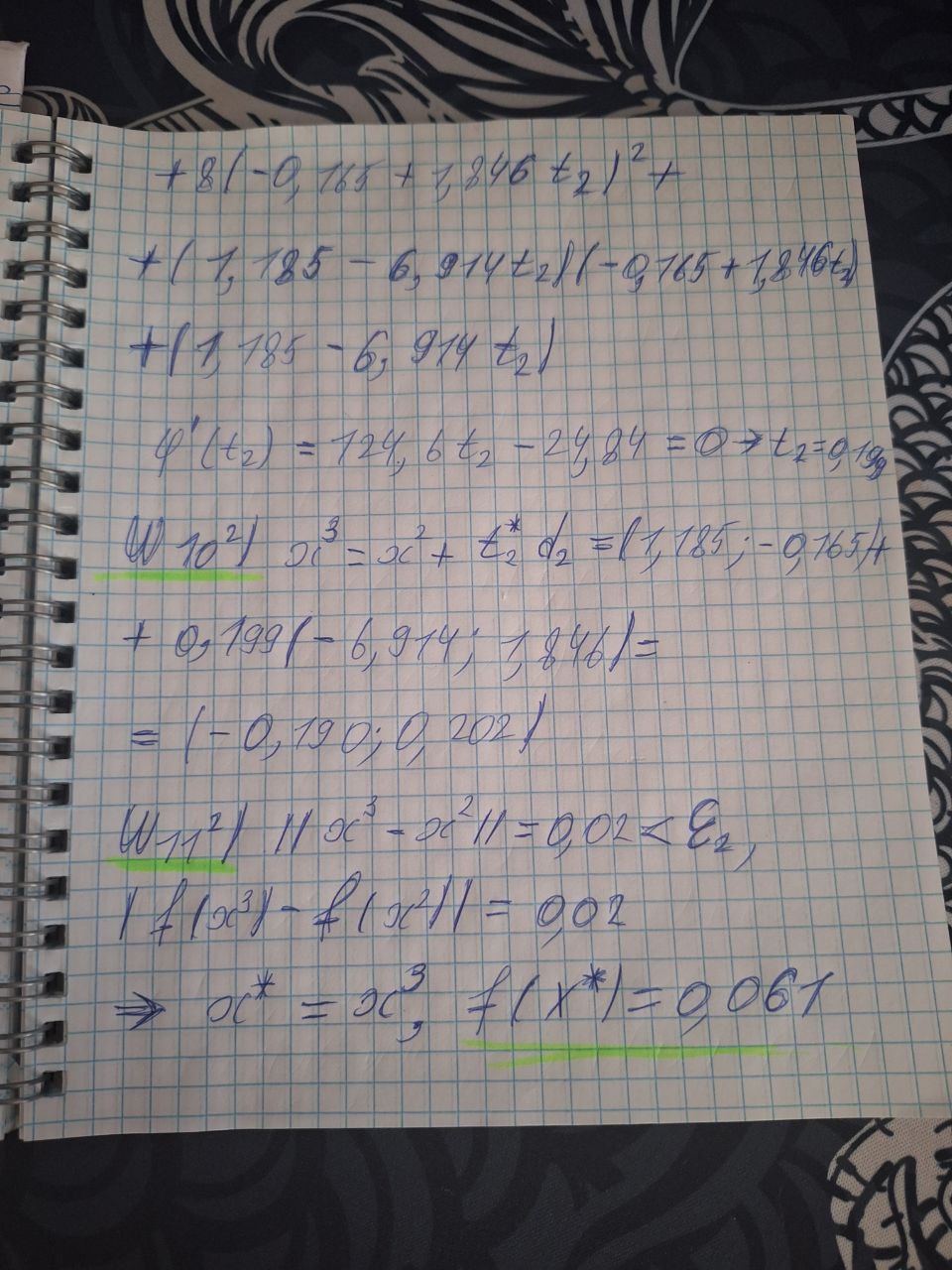
5.1 Пример работы алгоритма



****







5.2 Код программы

import math as mt

import numpy as np

import sympy as sp

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

def create\_x\_range() -> list[list[*float*]]:

    """

    Функция, которая создаёт диапазон для функции

    Args: отсутствует

    Return: список списков с точками

    """

    x\_range = []

    step = 10 / 500

    for i in range(1000):

        x\_range.append([-5 + (step \* i), -5 + (step \* i)])

    return x\_range

def f(*x\_list*: list[list[*float*]]) -> list[*float*]:

    """

    Функция двух переменных

    Args: x - список значений x

    Return: список значений функции

    """

    result = []

    for x in x\_list:

        result.append([x[0]\*\*2 + 8\*(x[1]\*\*2) + x[0]\*x[1] + x[0]])

    return result

def grad\_f(*x*: list[*float*]) -> list[*float*]:

    """

    Функция, которая считает градиент функции f

    Args: x - аргумент функции

    Return: список значений

    """

    return [2\*x[0] + x[1] + 1, 16\*x[1] + x[0]]

def norm\_grad(*grad*: list[*float*]) -> *float*:

    """

    Функция, которая считает норму градиента

    Args: grad - градиент функции

    Return: значение нормы градиента функции

    """

    return mt.sqrt(grad[0]\*\*2 + grad[1]\*\*2)

def calculate\_d(*x*: list[*float*], *betta*: *float*, *d\_0*: list[*float*], *flag*: *int*) -> list[*float*]:

    """

    Функция, которая считает параметр d

    Args: x - список значений x; betta - аргумент бетта; d\_0 - предыдущее значение d; flag - способ подсчёта нового d

    Return: новое значение d в виде списка

    """

    if flag == 1:

        return [-x for x in grad\_f(x)]

    elif flag == 2:

        return np.array([-x for x in grad\_f(x)]) + np.array(betta \* np.array(d\_0))

    else:

        print("Неверное значение флага! Флаг может быть 1 или 2!")

def calculate\_betta(*new\_norm\_grad*: list[*float*], *old\_norm\_grad*: list[*float*]) -> *float*:

    """

    Функция, которая считает аргумент бетта

    Args: grad\_new и grad\_old - новая и старая норма градиента (от нового и старого х)

    Return: аргумент бетта

    """

    return (new\_norm\_grad\*\*2) / (old\_norm\_grad\*\*2)

def calculate\_t(*x\_old*: list[*float*], *d*: list[*float*]) -> *float*:

    """

    Функция, которая вычисляет значение t на какой-то итерации

    Args: x\_old - список значений; d - коэффициент d

    Return: значение t

    """

    t = sp.symbols("t")  #объявляем символьную переменную

    expression\_x\_new = [x\_old[0] + t \* d[0], x\_old[1] + t \* d[1]]

    after\_f = (expression\_x\_new[0]\*\*2) + (8 \* (expression\_x\_new[1]\*\*2)) + (expression\_x\_new[0] \* expression\_x\_new[1]) + expression\_x\_new[0]

    after\_f\_derivative = sp.diff(after\_f, t)  #берём производную по t

    solution\_t = sp.solve(after\_f\_derivative, t)  #решение уравнения производной = 0 для t

    return solution\_t

def calculate\_x\_new(*x\_old*: list[*float*], *t*: *float*, *d*: list[*float*]) -> list[*float*]:

    """

    Функция, которая вычисляет новое значение вектора x

    Args: x\_old - предыдущее значение вектора x; t - коэффициент t; d - коэффициент d

    Return: новый вектор x

    """

    return np.array(x\_old) + t \* np.array(d)

def norm\_between\_x(*x\_old*: list[*float*], *x\_new*: list[*float*]) -> *float*:

    """

    Функция, которая вычисляет норму между новым и старым х

    Args: x\_old и x\_new - старый и новый х

    Return: значение нормы

    """

    return mt.sqrt((x\_new[0] - x\_old[0])\*\*2 + (x\_new[1] - x\_old[1])\*\*2)

def mod\_between\_f(*x\_old*: list[*float*], *x\_new*: list[*float*]) -> *float*:

    """

    Функция, которая вычисляет модуль между разницей новой и старой f

    Args: x\_old и x\_new - старый и новый х

    Return: значение модуля

    """

    return abs(f(x\_new)[0][0] - f(x\_old)[0][0])

# Создать списки для сохранения значений x и f(x)

x\_history = []

f\_history = []

x\_0 = [1.5, 0.5]

epsilon\_1 = 0.1

epsilon\_2 = 0.15

m = 10

k = 0

old\_norm\_grad = 0

d\_old = 0

flag = 0

while k < m:

    # Добавить текущее значение x и f(x) в историю

    x\_history.append(x\_0)

    f\_history.append(f([x\_0])[0][0])

    #шаг 3

    grad\_x\_0 = grad\_f(x\_0)

    #print(f"grad\_x\_0 = {grad\_x\_0}")

    #шаг 4

    norm\_grad\_x\_0 = norm\_grad(grad\_x\_0)

    if norm\_grad\_x\_0 < epsilon\_1:

        x\_min = x\_0

        f\_min = f([x\_min])

        break

    else:

        #шаг 5

        if k >= m:

            x\_min = x\_0

            f\_min = f([x\_min])

            break

        elif k == 0:

            #шаг 6

            d = calculate\_d(x\_0, None, None, 1)

        elif k > 0:

            #шаг 7

            betta = calculate\_betta(norm\_grad\_x\_0, old\_norm\_grad)

            #print(f"betta = {betta}")

            #шаг 8

            d = calculate\_d(x\_0, betta, d\_old, 2)

        #print(f"d = {d}")

        #шаг 9

        t = calculate\_t(x\_0, d)

        #print(f"t = {t}")

        #шаг 10

        x\_new = calculate\_x\_new(x\_0, t, d)

        #print(f"x\_new = {x\_new}")

        #шаг 11

        between\_x = norm\_between\_x(x\_0, x\_new)

        #print(f"between\_x = {between\_x}")

        between\_f = mod\_between\_f([x\_0], [x\_new])

        #print(f"between\_f = {between\_f}")

        if between\_x < epsilon\_2 and between\_f < epsilon\_2:

            if flag == 1:

                flag = 2

                x\_min = x\_new

                f\_min = f([x\_min])

                break

            else:

                flag = 1

                x\_0 = x\_new

                old\_norm\_grad = norm\_grad\_x\_0

                d\_old = d

                k += 1

        else:

            x\_0 = x\_new

            old\_norm\_grad = norm\_grad\_x\_0

            d\_old = d

            k += 1

print(f"Координаты точки минимума: {x\_min};\n f(x): {f\_min};\n k: {k}")

# Изменить часть для построения графика

x\_range = create\_x\_range()

x1\_values = np.array([x\_range[i][0] for i in range(len(x\_range))])

x2\_values = x1\_values.copy()

x, y = np.meshgrid(x1\_values, x2\_values)

z = np.array(f(x\_range))

fig = plt.figure()

ax = fig.add\_subplot(1, 1, 1, *projection*="3d")

ax.plot\_surface(x, y, z, *cmap*='viridis', *alpha*=0.6)

ax.scatter(x\_min[0], x\_min[1], f\_min, *color*="red")

# Добавить отображение шагов минимизации

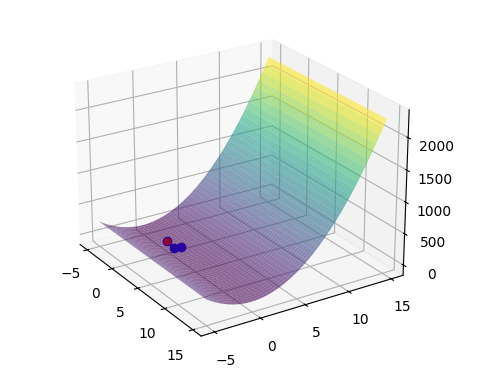
x\_history = np.array(x\_history)

f\_history = np.array(f\_history)

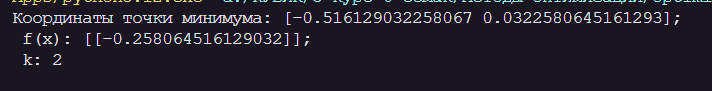
ax.plot(x\_history[:, 0], x\_history[:, 1], f\_history, *marker*='o', *color*='blue')

plt.show()

5.3 Вывод программы







**ВЫВОД**

Методы градиентного спуска, Ньютона, Ньютона-Рафсона и Флетчера-Ривса - разные стратегии оптимизации функций, нацеленные на поиск локальных экстремумов.

1) Градиентный спуск использует градиент функции для определения направления наискорейшего убывания переменных. Самый простой в реализации, но делает большее кол-во итераций.

2) Ньютона использует гессиан функции для определения направления минимизации.

3) Ньютона-Рафсона комбинирует обратный гессиан и градиент для коррекции направления.

Второй и третий работают похожим образом, используют гесениан для поиска минимума функции, подходят в схожих ситуациях.

4) Флетчера-Ривса использует предыдущие направления для определения нового. Каждый из них может быть эффективен в зависимости от характера функции, ее выпуклости и размерности. Точный, но требует больше вычислительных мощностей.