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 №2**

Работу выполнил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А. А. Иванов

(подпись)

Направление подготовки 02.03.03 Математическое обеспечение и администрирование информационных систем

Направленность Методы оптимизации

Руководитель А.С. Чёрная

(подпись)

Краснодар

2024

**Постановка задачи**

Требуется для функции , начальной точки , точности и , предельного числа итераций M = 10 и градиента функции найти точку минимума функции, значение функции в этой точке и значение k несколькими метолами (методом наискорейшего градиентного спуска, методом Ньютона, методом Ньютона-Рафсона, методом Флетчера-Ривса), а также провести сравнение методов.

**Метод наискорейшего градиентного спуска**

**Алгоритм метода**

Метод наискорейшего градиентного спуска - это итерационный алгоритм оптимизации, используемый для нахождения локального минимума (или максимума) функции. Он основан на идее использования градиента функции (вектора её частных производных) для определения направления наискорейшего убывания функции и последующего движения в этом направлении.

**Пример работы алгоритма**

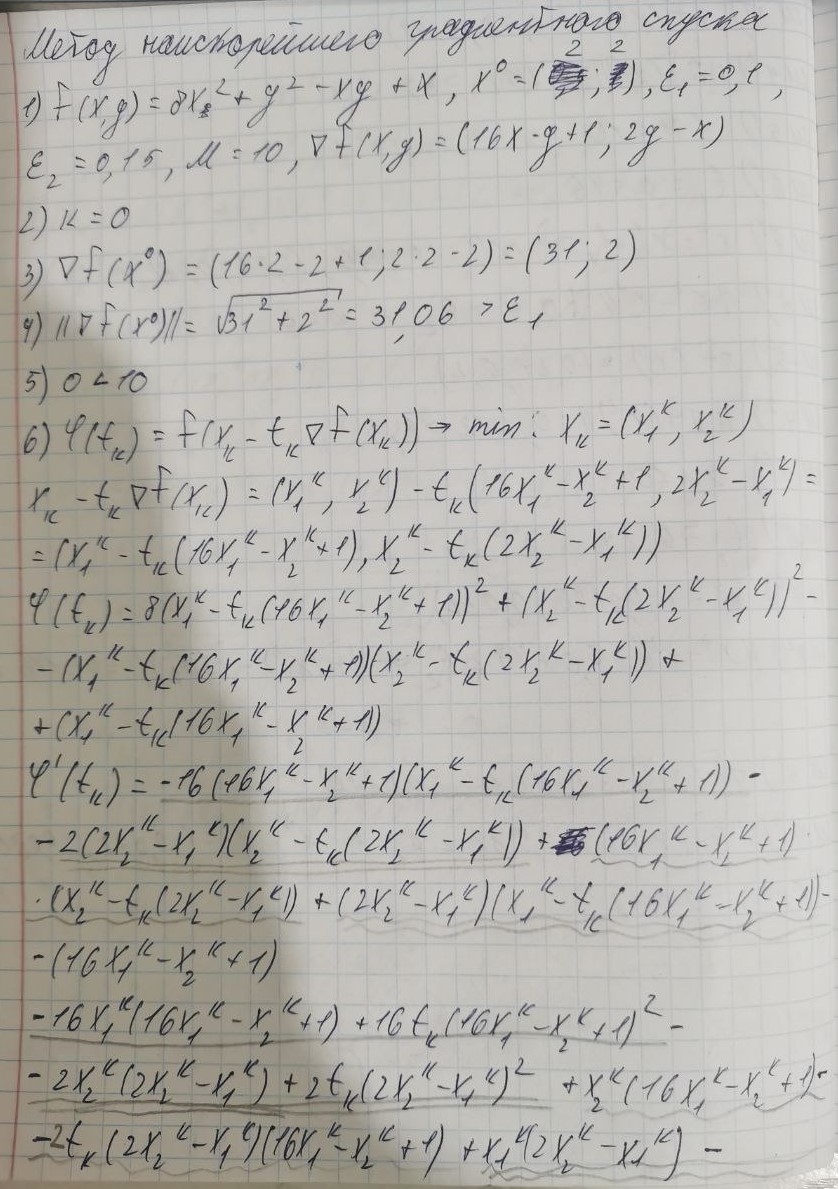


Рисунок 1 – пример работы метода наискорейшего градиентного спуска

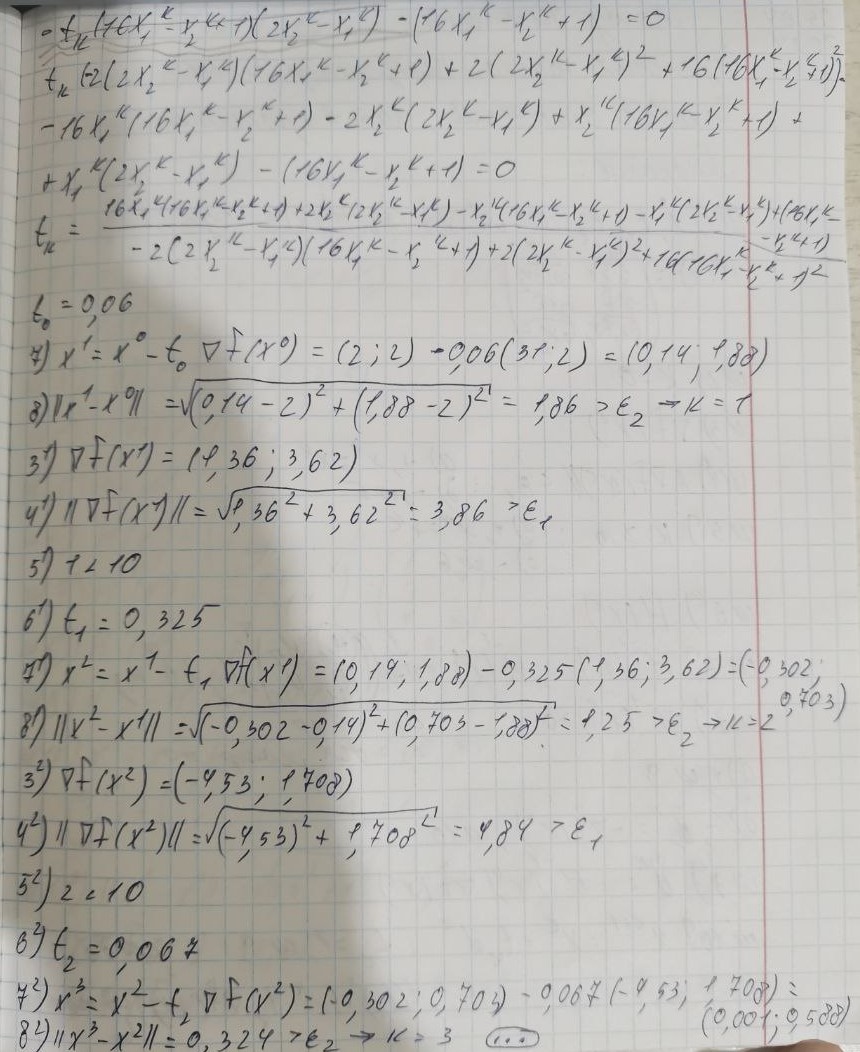


Рисунок 2 – пример работы метода наискорейшего градиентного спуска

**Код программы**

import numpy as np

import math as mt

import matplotlib.pyplot as plt

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([-10 + (step \* i), -10 + (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([(8 \* (x[0]\*\*2)) + (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

    """

    result = []

    result.append((16 \* *x*[0]) - *x*[1] + 1)

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

    return result

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

    """

    Функция, которая вычисляет значение t на какой-то итерации

    Args: x - аргумент

    Return: значение t

    """

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

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

    """

    new\_x = []

    new\_x.append(*x\_old*[0] - *t* \* grad\_f(*x\_old*)[0])

    new\_x.append(*x\_old*[1] - *t* \* grad\_f(*x\_old*)[1])

    return new\_x

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])

x\_0 = [2, 2]

epsilon\_1 = 0.1

epsilon\_2 = 0.15

m = 10

k = 0

flag = 0

while k < m:

*#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"x\_min: {x\_min}; f(x\_min): {f\_min}; k: {k}")

x\_range = create\_x\_range()

tmp\_x1 = [x\_range[i][0] for i in range(len(x\_range))]

tmp\_x2 = tmp\_x1.copy()

*#создаём сетку значений для построение поверхности*

x, y = np.meshgrid(tmp\_x1, tmp\_x2)

*#превращение массива от функции f в массив numpy (нужно для lot\_surface)*

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

fig = plt.figure()

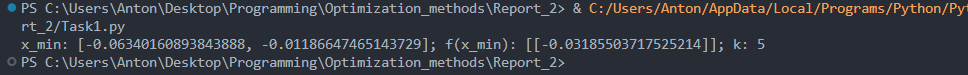
ax = fig.add\_subplot(1, 1, 1, *projection*="3d")

ax.plot\_surface(x, y, z)

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

plt.show()

**Вывод программы**



**Метод Ньютона**

**Алгоритм метода**

Метод Ньютона, это итерационный численный метод оптимизации, используемый для нахождения локального минимума (или максимума) функции. Он основан на сходимости в окрестности точки минимума.

**Пример работы алгоритма**

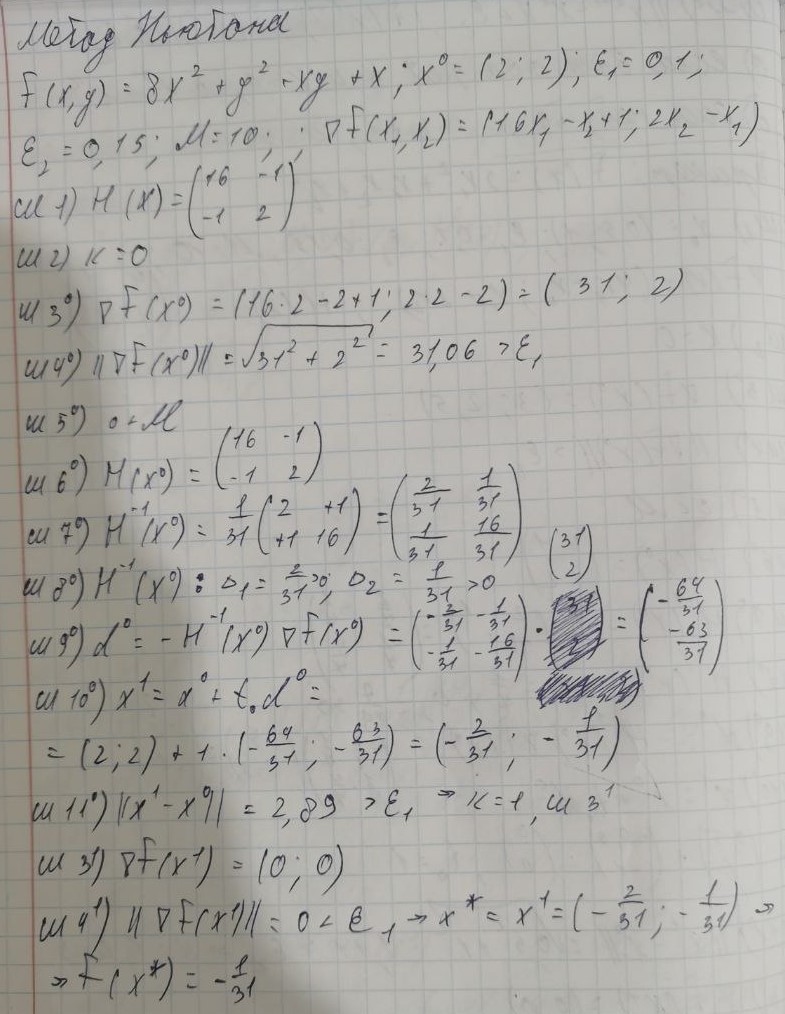
****

Рисунок 3 – пример работы метода Ньютона

**Код программы**

import math as mt

import numpy as np

import matplotlib.pyplot as plt

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([-10 + (step \* i), -10 + (step \* i)])

    return x\_range

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

    """

    Функция двух переменных

    Args: x - список значений x

    Return: список значений функции

    """

    result = []

    for x in *x\_list*:

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

    """

    result = []

    result.append((16 \* *x*[0]) - *x*[1] + 1)

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

    return result

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

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])

x\_0 = [2, 2]

epsilon\_1 = 0.1

epsilon\_2 = 0.15

m = 10

k = 0

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

flag = 0

while k < m:

*#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"x\_min: {x\_min}; f(x\_min): {f\_min}; k: {k}")

x\_range = create\_x\_range()

tmp\_x1 = [x\_range[i][0] for i in range(len(x\_range))]

tmp\_x2 = tmp\_x1.copy()

*#создаём сетку значений для построение поверхности*

x, y = np.meshgrid(tmp\_x1, tmp\_x2)

*#превращение массива от функции f в массив numpy (нужно для lot\_surface)*

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

fig = plt.figure()

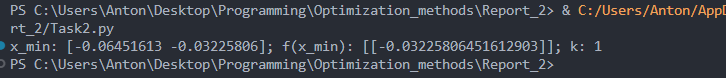
ax = fig.add\_subplot(1, 1, 1, *projection*="3d")

ax.plot\_surface(x, y, z)

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

plt.show()

**Вывод программы**

****

**Метод Ньютона-Рафсона**

**Алгоритм метода**

Метод Ньютона-Рафсона — это итерационный численный метод оптимизации, используемый для нахождения локального минимума (или максимума) функции. Он основан на аппроксимации функции квадратичным многочленом в окрестности текущей точки и нахождении минимума этого многочлена.

**Пример работы алгоритма**

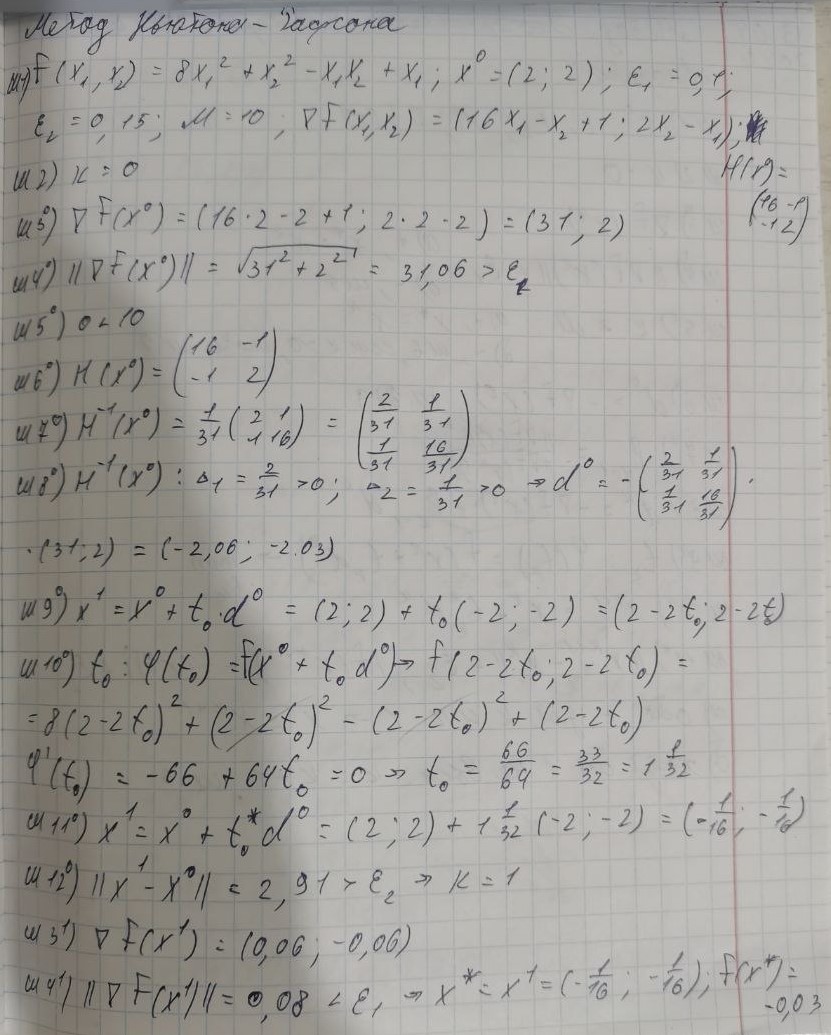
****

Рисунок 4 – пример работы метода Ньютона-Рафсона

**Код программы**

import math as mt

import numpy as np

import sympy as sp

import matplotlib.pyplot as plt

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([-10 + (step \* i), -10 + (step \* i)])

    return x\_range

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

    """

    Функция двух переменных

    Args: x - список значений x

    Return: список значений функции

    """

    result = []

    for x in *x\_list*:

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

    """

    result = []

    result.append((16 \* *x*[0]) - *x*[1] + 1)

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

    return result

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

x\_0 = [2, 2]

epsilon\_1 = 0.1

epsilon\_2 = 0.15

m = 10

k = 0

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

flag = 0

while k < m:

*#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"x\_min: {x\_min}; f(x\_min): {f\_min}; k: {k}")

x\_range = create\_x\_range()

tmp\_x1 = [x\_range[i][0] for i in range(len(x\_range))]

tmp\_x2 = tmp\_x1.copy()

*#создаём сетку значений для построение поверхности*

x, y = np.meshgrid(tmp\_x1, tmp\_x2)

*#превращение массива от функции f в массив numpy (нужно для lot\_surface)*

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

fig = plt.figure()

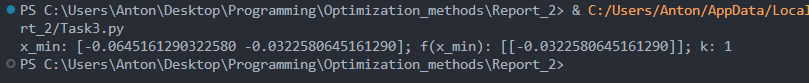
ax = fig.add\_subplot(1, 1, 1, *projection*="3d")

ax.plot\_surface(x, y, z)

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

plt.show()

**Вывод программы**

****

**Метод Флетчера-Ривса**

**Алгоритм метода**

Метод Флетчера-Ривса - это алгоритм оптимизации безусловной оптимизации, который использует только значения функции и её градиент для поиска минимума. Он относится к классу методов оптимизации сопряженных направлений.

**Пример работы алгоритма**

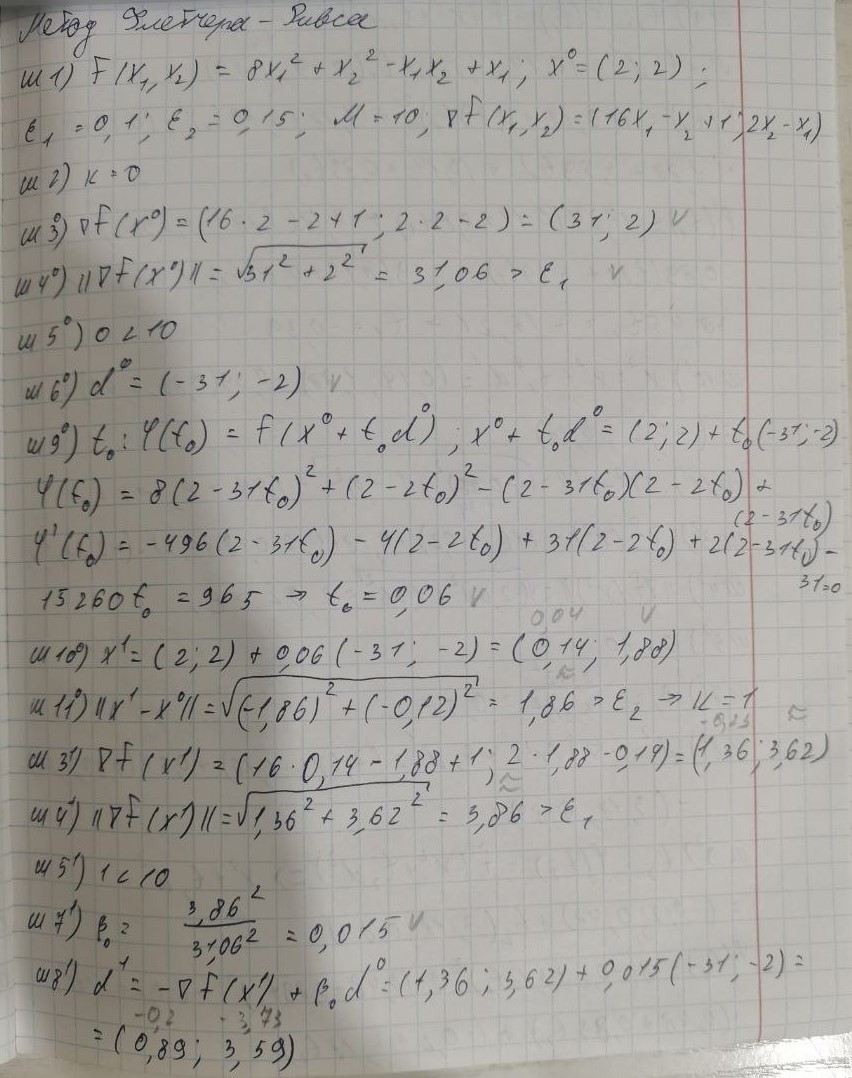
****

Рисунок 5 – пример работы алгоритма Флетчера-Ривса

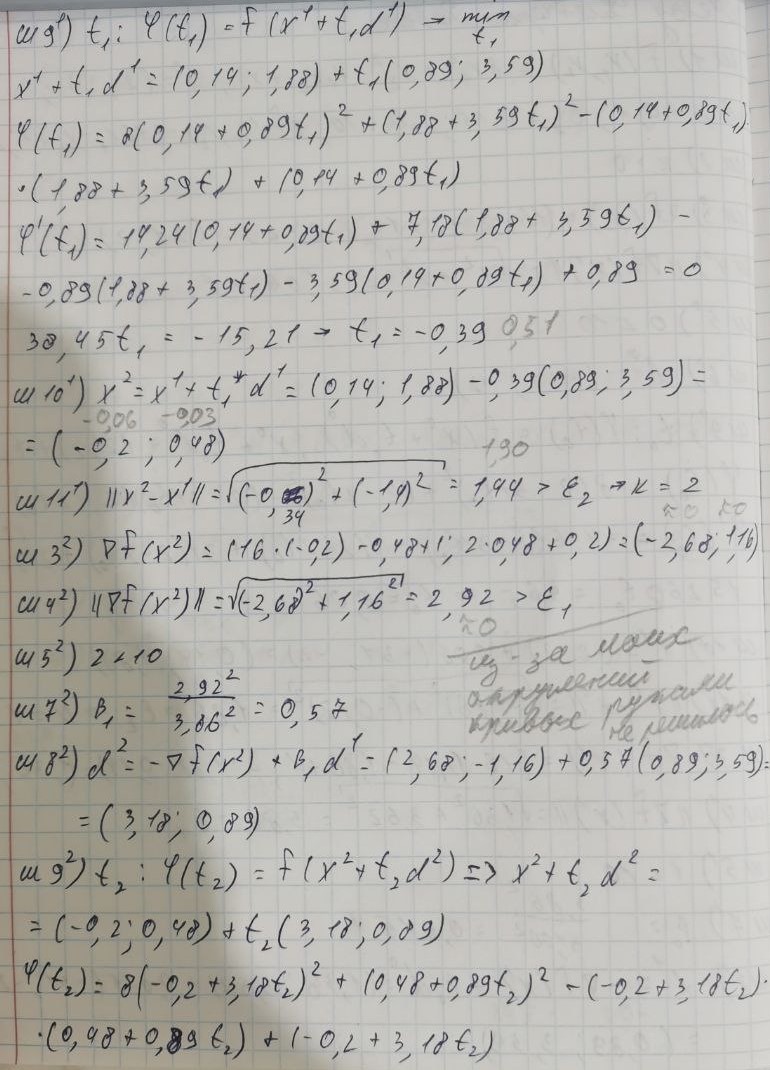


Рисунок 6 – пример работы алгоритма Флетчера-Ривса

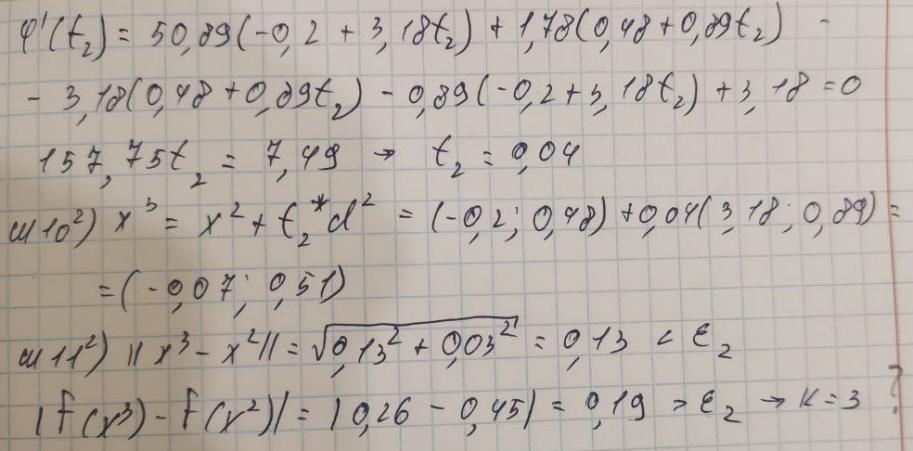


Рисунок 7 – пример работы алгоритма Флетчера-Ривса

**Код программы**

import math as mt

import numpy as np

import sympy as sp

import matplotlib.pyplot as plt

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([-10 + (step \* i), -10 + (step \* i)])

    return x\_range

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

    """

    Функция двух переменных

    Args: x - список значений x

    Return: список значений функции

    """

    result = []

    for x in *x\_list*:

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

    """

    result = []

    result.append((16 \* *x*[0]) - *x*[1] + 1)

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

    return result

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

x\_0 = [2, 2]

epsilon\_1 = 0.1

epsilon\_2 = 0.15

m = 10

k = 0

old\_norm\_grad = 0

d\_old = 0

flag = 0

all\_x\_0 = [x\_0]

while k < m:

    print(f"k = {k}")

*#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)

    print(f"norm\_grad\_x\_0 = {norm\_grad\_x\_0}")

    if norm\_grad\_x\_0 < epsilon\_1:

        x\_min = x\_0

        f\_min = f([x\_min])

        all\_x\_0.append(x\_min)

        break

    else:

*#шаг 5*

        if k >= m:

            x\_min = x\_0

            f\_min = f([x\_min])

            all\_x\_0.append(x\_min)

            break

        elif k == 0:

*#шаг 6*

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

        elif k > 0 and k < 10:

*#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])

                all\_x\_0.append(x\_min)

                break

            else:

                flag = 1

                x\_0 = x\_new

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

                d\_old = d

                k += 1

                all\_x\_0.append(x\_new)

        else:

            x\_0 = x\_new

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

            d\_old = d

            k += 1

            all\_x\_0.append(x\_new)

print(f"x\_min: {x\_min}; f(x\_min): {f\_min}; k: {k}")

all\_x\_0.pop()

all\_f\_x\_0 = [f([x]) for x in all\_x\_0]

x\_range = create\_x\_range()

tmp\_x1 = [x\_range[i][0] for i in range(len(x\_range))]

tmp\_x2 = tmp\_x1.copy()

*#создаём сетку значений для построение поверхности*

x, y = np.meshgrid(tmp\_x1, tmp\_x2)

*#превращение массива от функции f в массив numpy (нужно для lot\_surface)*

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

fig = plt.figure()

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

ax.plot\_surface(x, y, z)

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

for i in range(len(all\_x\_0)):

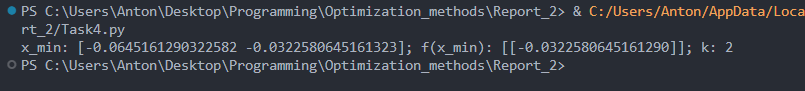
    ax.scatter(all\_x\_0[i][0], all\_x\_0[i][1], all\_f\_x\_0[i][0][0], *color*="red")

    if i == len(all\_x\_0) - 1:

        ax.text(all\_x\_0[i][0], all\_x\_0[i][1], all\_f\_x\_0[i][0][0], "Минимальное значение функции", *color*="black", *fontsize*=10, *ha*="right")

plt.show()

**Вывод программы**

****

**График функции и точек**

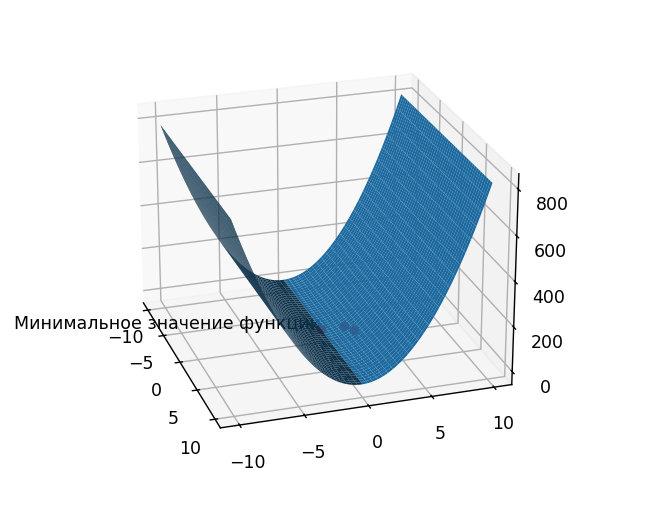
****

Рисунок 8 – график построенной функции и точек

**Сравнение методов**

1. Метод наискорейшего градиентного спуска:

Основан на использовании градиента функции (первой производной);

Каждый шаг делается в направлении, противоположном градиенту, чтобы минимизировать функцию;

Прост в реализации, но может иметь медленную сходимость на сложных функциях.

1. Метод Ньютона:

Использует информацию о второй производной (гессиане) функции;

Обеспечивает быструю сходимость в окрестности минимума, особенно квадратичных функций;

Может столкнуться с проблемами неустойчивости или медленной сходимостью на сложных функциях из-за обращения гессиана;

Требует вычислительно затратных операций обращения матрицы.

1. Метод Ньютона-Рафсона:

Это модификация метода Ньютона, где используется приближенная обратная гессиана;

Это может быть выгодно на практике, особенно если обращение гессиана слишком сложно или затратно;

Обычно обладает быстрой сходимостью и хорошей производительностью на широком классе функций.

1. Метод Флетчера-Ривса:

Это метод оптимизации сопряженных градиентов;

Он использует информацию о градиентах в разных направлениях для ускорения сходимости;

Обычно хорошо работает на функциях с вытянутыми или сложными линиями уровня;

Может иметь проблемы с медленной сходимостью на невыпуклых функциях.

С точки зрения кода самым оптимальным методом показался метод Ньютона-Рафсона, так как он как и метод Ньютона находит точку минимума всего за 2 итерации, но немного проще в реализации. Метод наискорейшего градиентного спуска показал себя как самый долгий метод, так как совершает около 6 итераций для нахождения той же точки минимума.