Министерство науки и высшего образования Российской Федерации
федеральное государственное автономное образовательное учреждение высшего образования
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчет

по лабораторной работе №3 «Методы спуска»

по дисциплине «**Прикладная математика**»

Автор: Константинова Ольга Алексеевна, Векинцева Виктория Александровна

Факультет: Информационные технологии и программирование

Группа: M32111

УНИВЕРСИТЕТ ИТМО

Санкт-Петербург, 2023

Цель работы: изучение методов поиска минимума целевой функции - метода градиентного спуска с постоянным и переменным шагом, метода наискорейшего спуска и метода сопряженных градиентов.

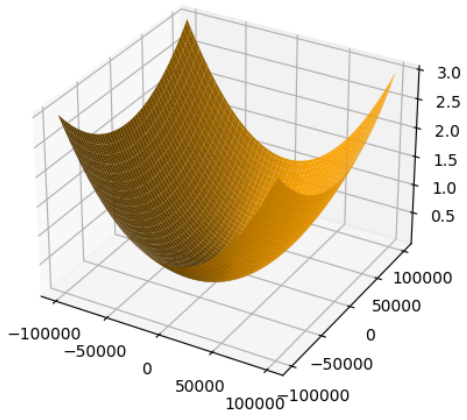
Задачи:

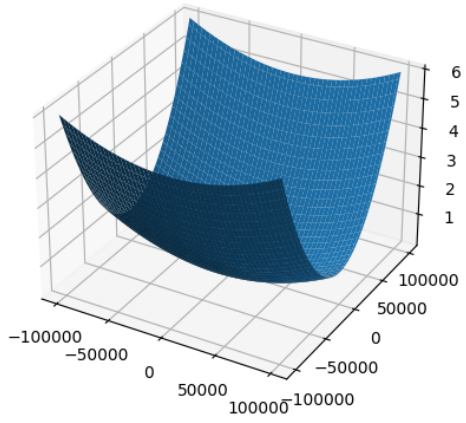
1. Реализовать алгоритмы градиентного спуска с постоянным шагом, спуска с дроблением шага, используя условие Армихо, метода наискорейшего спуска и метода сопряженных градиентов.
2. Проанализировать траектории предложенных алгоритмов на примере квадратичных функций от двух переменных, на которых работа методов будет отличаться.
3. Для каждой функции:
    a) исследовать сходимость градиентного спуска с постоянным шагом и сравнить полученные результаты для выбранных функций;
    b) сравнить эффективность методов с точки зрения количества вычислений минимизируемой функции и её градиентов;
    c) исследовать работу методов в зависимости от выбора начальной точки;
    d) в каждом случае построить графики с линиями уровня и траекториями метода.

Ход работы:

Для выполнения данной лабораторной работы были изучены следующие квадратичные функции:

$$F_1(x_1, x_2) = 2 \cdot x^2 + (y-3)^2 \qquad F_2(x_1, x_2) = (x-1)^2 + 5 \cdot y^2 + 2 \cdot x$$





Метод градиентного спуска с постоянным шагом

Выбираем начальную точку x_0 и вычисляем антиградиент в данной точке.

Антиградиент – вектор, который показывает направление наискорейшего уменьшения функции в данной точке.

Делаем шаг с выбранной постоянной длиной на луче, определяемым вектором антиградиента для того, чтобы вычислить x_1.
Повторяем процедуру – вычисление антиградиента в очередной точке и выполнение шага с фиксированной длиной – до тех пор, пока не выполнится критерий завершения.

Критерием является неравенство $\| -\nabla F(x_1,\ x_2) \|^2 \leq \varepsilon$ – квадрат нормы антиградиента в точке должен стать меньше заданной точности.

Норма вектора – длина вектора.

Для расчёта антиградиента требуется две частные производные по x_1 и x_2, вычислить их значение в текущей точке итерации:

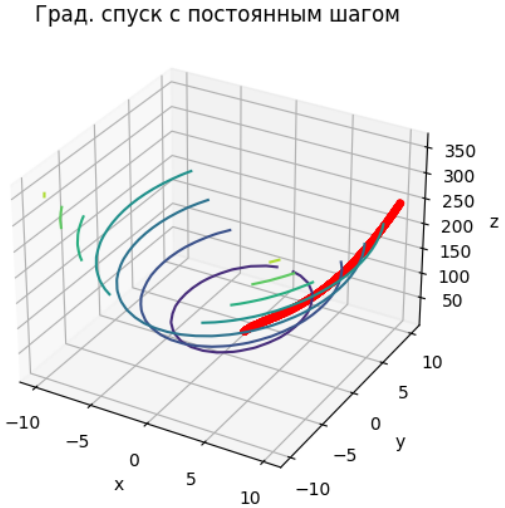
$$-\nabla F(x_1, x_2) = (-\frac{dF}{dx_1}, -\frac{dF}{dx_2})$$

Для первой функции:

Заданная точность	Шаг	Начальная точка x_0	Точка минимума	Значение минимума	Количество итераций
0.001	0.1	(10, 10)	Алгоритм не сошёлся		
0.001	0.01	(10, 10)	(-0.00000430, 3.00699947)	0.00004899	1277
0.0001	0.001	(10, 10)	(0.00000402, 3.00496328)	0.00002463	12762

Алгоритм показывает результат, когда шаг достаточно маленький. Также начальная точка должна находиться достаточно близко к искомой точке минимума, чтобы количество итераций не превысило порог (в нашей реализации порог – 1 млн шагов).

Траектория алгоритма для первой функции выглядит следующим образом:

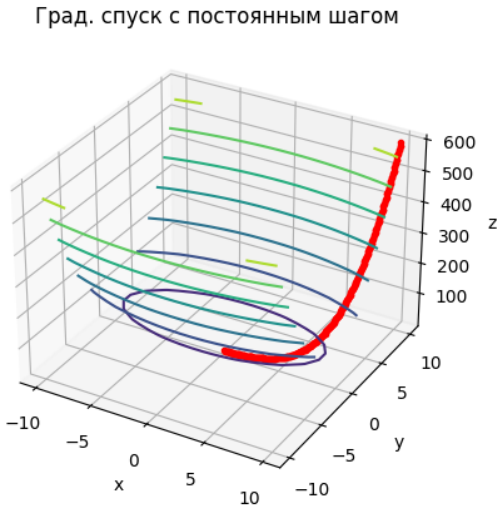


Для второй функции:

Заданная точность	Шаг	Начальная точка x_0	Точка минимума	Значение минимума	Количество итераций
0.001	0.1	(10, 10)	Алгоритм не сошёлся		
0.001	0.01	(10, 10)	(0.00772145, 0.00000000)	1.00005962	1641
0.0001	0.001	(10, 10)	(0.00480987, 0.00000000)	1.00002313	16402

Результаты, в целом, аналогичны тому, что мы видели в случае первой функции.

Траектория алгоритма:



Поверхность второй функции отличается от первой, из чего видно, что траектория на начальном этапе далека от оптимальной, потому что движение происходит не точно в сторону минимума (не кратчайшим путём).

Реализация метода на языке программирования python:

```
def const_descent(e, h, x0, func, part_deriv_x, part_deriv_y):
    print("Градиентный спуск с постоянным шагом с e = {:0.8f}, h = {:0.8f}, начало в
{}".format(e, h, x0))
    ax = plt.figure().add_subplot(projection='3d')
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("z")
    x_show, y_show = np.meshgrid(np.linspace(-10, 10, 20), np.linspace(-10, 10, 20))
    z_show = func(x_show, y_show)
    ax.contour(x_show, y_show, z_show)
    iterations = 0  # количество итераций
    x = x0
    fx = func(x[0], x[1])
    while True:
        iterations += 1
        if iterations > 1000000:
            print("Алгоритм не сошёлся!")
            break
        ag = antigrad(x[0], x[1], part_deriv_x, part_deriv_y)  # считаем антиградиент в текущей
точке (вектор)
        ag_norm = np.sqrt(ag[0] ** 2 + ag[1] ** 2)  # вычисляем норму (длину) вектора
антиградиента
        if ag_norm ** 2 < e:
            fx = func(x[0], x[1])  # вычисляем значение ф-и в новой точке
            print("Минимум в точке: [{:0.8f}, {:0.8f}], значение минимума: {:0.8f}, всего
итераций: {}"
                  .format(x[0], x[1], fx, iterations))
            break
        ag_normalized = [ag[0] / ag_norm, ag[1] / ag_norm]  # получаем единичный вектор,
направленный как вектор антиградиент
        x_prev = x
```

```
        fx_prev = fx
        x = [ag_normalized[0] * h + x[0], ag_normalized[1] * h + x[1]]  # делаем шаг в
направлении убывания длиной h
        fx = func(x[0], x[1])
        ax.plot([x_prev[0], x[0]], [x_prev[1], x[1]], [fx_prev, fx], '.-', c='red')

    plt.title("Град. спуск с постоянным шагом")
    plt.show()
```

Метод градиентного спуска с переменным шагом

На каждой итерации повторяем те же действия, что и в предыдущем методе – вычисляем антиградиент в очередной точке. Шаг для вычисления следующей точки при движении вдоль антиградиента выбирается более оптимальным способом – используя условие Армихо.

Для этого будем рассматривать вспомогательную функцию, зависящую от h:

$$\varphi_k(h) = F(x_k + h \cdot d_k)$$

(где $d_k = -\nabla F(x_k)$ – антиградиент в рассматриваемой точке, x_k – это вектор, заданный двумя числами – координатами рассматриваемой точки на данной итерации)

Производная этой функции $\varphi_k'(h) = (\nabla F(x_k + h \cdot d_k),\ d_k)$ – скалярное произведение антиградиента и градиента, вычисленного в точке с шагом h, лежащей на луче антиградиента.

Условие Армихо для выбора h:

$$\varphi_k(h) \leq \varphi_k(0) + c \cdot h \cdot \varphi_k'(0)$$

Алгоритм для выбора h:

1. Выбираем константу c (в нашей реализации выбрана 0.001)
2. Полагаем h равным какому-то предельному значению (например, 100)
3. Начинаем цикл, уменьшающий h в 2 раза, продолжающийся до тех пор, пока не выполнится условие Армихо

Вычисленный шаг h используем для определения следующей точки метода:

$$x_{k+1} = x_k + h \cdot d_k$$

т. е. откладываем шаг h от точки x_k в направлении антиградиента.

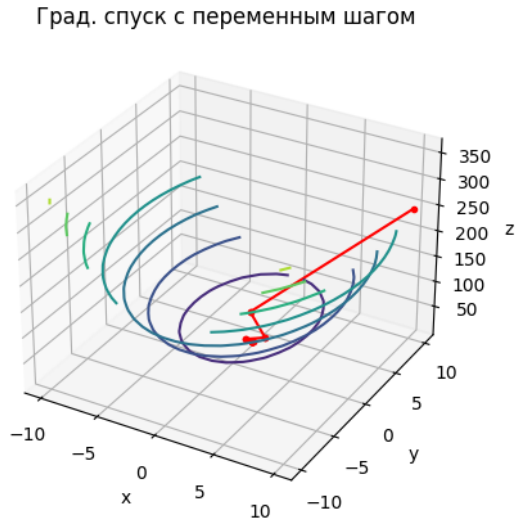
Метод продолжается до тех пор, пока не выполнится критерий завершения:
$\| -\nabla F(x_1,\ x_2) \|^2 \leq \varepsilon$ – квадрат нормы антиградиента в точке должен стать меньше заданной точности.

Для первой функции получаем следующие результаты:

Заданная точность	Коэффициент c	Начальная точка x_0	Точка минимума	Значение минимума	Количество итераций
0.1	0.001	(10, 10)	(-0.04094427, 3.00171138)	0.00335580	6
0.01	0.001	(10, 10)	(0.00787320, 3.00069115)	0.00012445	7

0.00001	0.001	(10, 10)	(0.00025480, 3.00000262)	0.00000013	11

Траектория алгоритма:

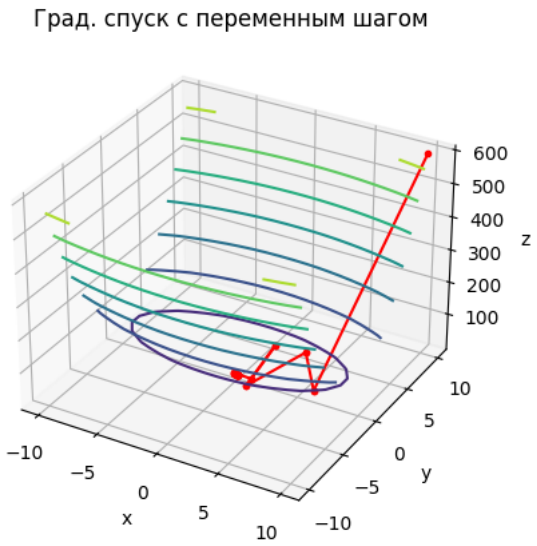


Сразу бросается в глаза, что у этого метода значительно меньшее количество итераций и с увеличением точности, они растут чрезвычайно медленно.

Для второй функции:

Заданная точность	Коэффициент c	Начальная точка x_0	Точка минимума	Значение минимума	Количество итераций
0.1	0.001	(10, 10)	(-0.10136549, -0.02016503)	1.01230810	10
0.01	0.001	(10, 10)	(-0.02611230, 0.00191688)	1.00070022	13
0.0001	0.001	(10, 10)	(-0.00112227, 0.00018369)	1.00000143	17

Траектория алгоритма:



Заметно, что выбор точки x_1 делается не оптимально – движение происходит в сторону от направления к точке минимума, как и в предыдущем методе.

Реализация метода:

```
def shrinking_step_descent(e, x0, func, part_deriv_x, part_deriv_y):
    print("Градиентный спуск с переменным шагом с e = {:0.8f}, c = 0.1, начало в {}".format(e,
x0))
    ax = plt.figure().add_subplot(projection='3d')
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("z")
    x_show, y_show = np.meshgrid(np.linspace(-10, 10, 20), np.linspace(-10, 10, 20))
    z_show = func(x_show, y_show)
    ax.contour(x_show, y_show, z_show)
    iterations = 0  # количество итераций
    x = x0
    fx = func(x[0], x[1])
    while True:
        iterations += 1
        if iterations > 1000000:
            print("Алгоритм не сошёлся!")
            break
        ag = antigrad(x[0], x[1], part_deriv_x, part_deriv_y)  # считаем антиградиент в текущей
точке (вектор)
        ag_norm = np.sqrt(ag[0] ** 2 + ag[1] ** 2)  # вычисляем норму (длину) вектора
антиградиента
        if ag_norm ** 2 < e:
            fx = func(x[0], x[1])  # вычисляем значение ф-и в новой точке
            print("Минимум в точке: [{:0.8f}, {:0.8f}], значение минимума: {:0.8f}, всего
итераций: {}"
                  .format(x[0], x[1], fx, iterations))
            break
        ag_normalized = [ag[0] / ag_norm, ag[1] / ag_norm]  # получаем единичный вектор,
направленный как вектор антиградиент
        fx = func(x[0], x[1])  # значение функции в текущей точке

        # производная функции "фи" = F(xk + h * dk), где xk - текущая точка (вектор),
        # dk - направление спуска для текущей точки (вектор), для h = 0
        # производную считаем как скалярное произведение антиградиента и направления спуска
        deriv_fi_0 = (-ag[0] * ag_normalized[0]) + (-ag[1] * ag_normalized[1])
        c = 0.001  # постоянная для условия армихо
        h = 100  # начальное значение шага, которое будет уменьшаться делением пополам
        # значение целевой функции для текущего шага h
        fi_h = func(x[0] + h * ag_normalized[0], x[1] + h * ag_normalized[1])
        while fi_h > fx + c * h * deriv_fi_0:
            h = h * 0.5
            fi_h = func(x[0] + h * ag_normalized[0], x[1] + h * ag_normalized[1])  # рассчитаем
значение целевой ф-и для нового h
        x_prev = x
        fx_prev = fx
        x = [ag_normalized[0] * h + x[0], ag_normalized[1] * h + x[1]]  # делаем шаг в
направлении убывания длиной h
        fx = func(x[0], x[1])
        ax.plot([x_prev[0], x[0]], [x_prev[1], x[1]], [fx_prev, fx], '.-', c='red')
    plt.title("Град. спуск с переменным шагом")
    plt.show()
```

Метод наискорейшего спуска

В этом методе на каждой итерации повторяем те же действия, что и в предыдущем методе – вычисляем антиградиент в очередной точке. Шаг для вычисления следующей точки при движении вдоль антиградиента выбирается более оптимальным способом – методом золотого сечения, примененным к вспомогательной функции, зависящей от h:

$$\varphi_k(h) = F(x_k + h \cdot d_k)$$

(где $d_k = -\nabla F(x_k)$ – антиградиент в рассматриваемой точке, x_k – это вектор, заданный двумя числами – координатами рассматриваемой точки на данной итерации)

Нашей целью является поиск значения h, в котором эта функция принимает минимальное значение на интервале допустимых значений для h (в нашем случае [0, 100]).

Поиск делаем при помощи золотого сечения.

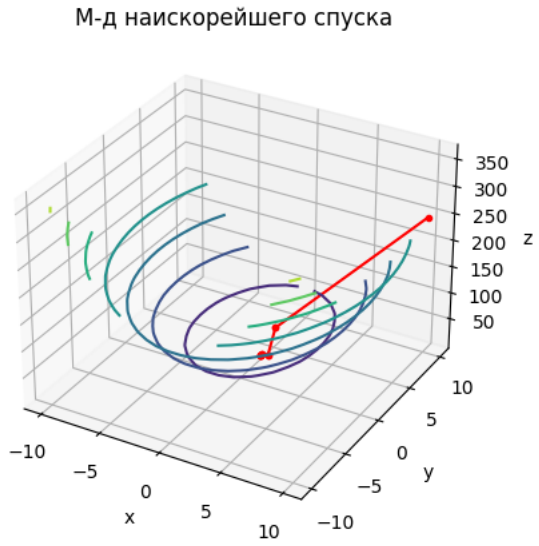
Метод продолжается до тех пор, пока не выполнится критерий завершения:
$\| -\nabla F(x_1,\ x_2) \|^2 \leq \varepsilon$ – квадрат нормы антиградиента в точке должен стать меньше заданной точности.

Для первой функции:

Заданная точность	Начальная точка x_0	Точка минимума	Значение минимума	Количество итераций
0.1	(10, 10)	(0.02736097, 3.01384514)	0.00168893	5
0.001	(10, 10)	(-0.00124748, 3.00700473)	0.00005218	6
0.00001	(10, 10)	(-0.00005773, 3.00032772)	0.00000011	8

Количество в этом методе ещё меньше, чем в методе с переменным шагом.

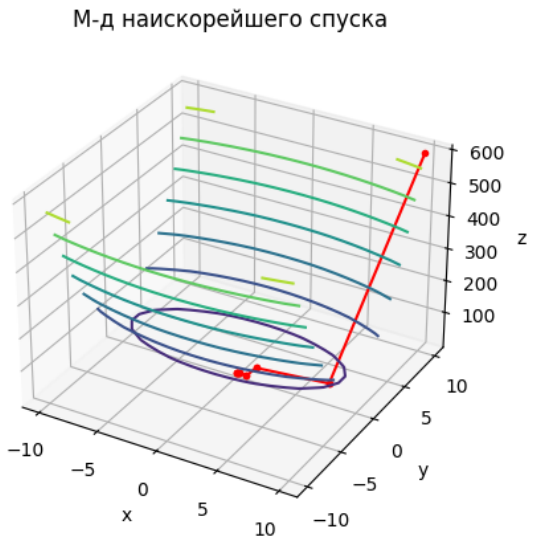
Траектория алгоритма:



Для второй функции:

Заданная точность	Начальная точка x_0	Точка минимума	Значение минимума	Количество итераций
0.1	(10, 10)	(0.00629752, 0.00718425)	1.00029773	7
0.001	(10, 10)	(0.00992127, -0.00050642)	1.00009971	8
0.00001	(10, 10)	(0.00099417, -0.00003759)	1.00000100	10

Траектория алгоритма:



Из графика видно, что траектория после первого шага более оптимальна в сравнении с траекторией для метода с переменным шагом.

Реализация метода:

```
def steepest_descent(e, x0, func, part_deriv_x, part_deriv_y, antigrad_func):
    print("Метод наискорейшего спуска с e = {:0.8f}, начало в {}".format(e, x0))
    ax = plt.figure().add_subplot(projection='3d')
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("z")
    x_show, y_show = np.meshgrid(np.linspace(-10, 10, 20), np.linspace(-10, 10, 20))
    z_show = func(x_show, y_show)
    ax.contour(x_show, y_show, z_show)
    iterations = 0  # количество итераций
    x = x0
    fx = func(x[0], x[1])
    while True:
        iterations += 1
        if iterations > 1000000:
            print("Алгоритм не сошёлся!")
            break
        ag = antigrad(x[0], x[1], part_deriv_x, part_deriv_y)  # считаем антиградиент в текущей
точке (вектор)
        ag_norm = np.sqrt(ag[0] ** 2 + ag[1] ** 2)  # вычисляем норму (длину) вектора
антиградиента
        if ag_norm ** 2 < e:
            fx = func(x[0], x[1])  # вычисляем значение ф-и в новой точке
            print("Минимум в точке: [{:0.8f}, {:0.8f}], значение минимума: {:0.8f}, всего
итераций: {}"
                  .format(x[0], x[1], fx, iterations))
            break
        ag_normalized = [ag[0] / ag_norm, ag[1] / ag_norm]  # получаем единичный вектор,
направленный как вектор антиградиент
        h = golden_cut(antigrad_func, x[0], x[1], ag_normalized, e, 0, 100)
        x_prev = x
        fx_prev = fx
        x = [ag_normalized[0] * h + x[0], ag_normalized[1] * h + x[1]]  # делаем шаг в
направлении убывания длиной h
        fx = func(x[0], x[1])
        ax.plot([x_prev[0], x[0]], [x_prev[1], x[1]], [fx_prev, fx], '.-', c='red')
    plt.title("М-д наискорейшего спуска")
    plt.show()
```

Метод сопряженных градиентов

Для первого шага вычисляем антиградиент для точки x_0 и находим шаг h, минимизируя при помощи золотого сечения вспомогательную функцию φ_1, зависящую от h:

$$\varphi_1(h) = F(x_0 + h \cdot d_1)$$

(где $d_1 = -\nabla F(x_0)$ – антиградиент в рассматриваемой точке, x_0 – это вектор, заданный двумя числами – координатами рассматриваемой точки на данной итерации)

Вычисляем точку x_1, делая шаг длиной h в направлении антиградиента от точки x_0.

Пусть вектор p_1 равен вектору d_1.

На каждом k-ом шаге выполняем следующие действия:

1. Находим антиградиент для точки x_k
2. Находим вектор p_k, сопряженный* с вектором p_{k-1}
3. Находим минимум вспомогательной функции $\varphi_k(h) = F(x_k + h \cdot p_k)$, используя метод золотого сечения для интервала для h от 0 до 100 включительно
4. Вычисляем точку x_{k+1}, делая шаг h в направлении вектора p_k от точки x_k

*Сопряженными векторами p_i и p_j называются вектора, для которых выполняется условие: $p_i^T \cdot H \cdot p_j = 0$, где H – матрица Гессе функционала F.

Для нашего случая квадратичного функционала F мы воспользуемся оптимизацией Флетчера-Ривса - в п.2 для вычисления сопряженного вектора p_k воспользуемся формулой:

$$p_k = d_k + w \cdot p_{k-1}$$

(где d_k – антиградиент из п.1, w – вычисляется по следующей формуле:

$$w = \frac{(d_k, d_k)}{(d_{k-1}, d_{k-1})}$$

т. е. отношение скалярного произведения антиградиента d_k на себя к скалярному произведению антиградиента d_{k-1} с предыдущей итерации на себя)

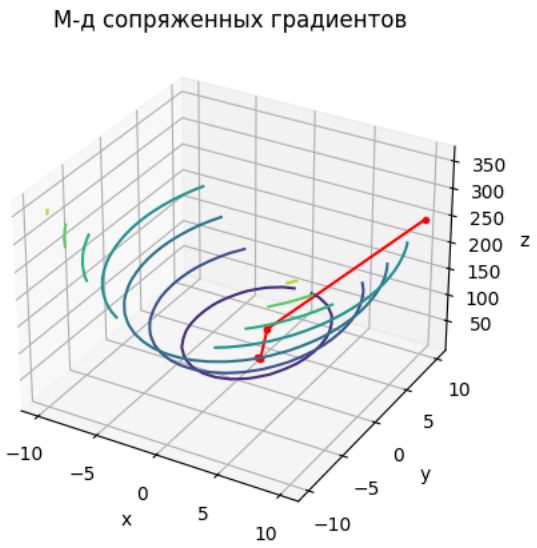
Алгоритм завершается, когда выполняется условие: $\| -\nabla F(x_1, x_2) \|^2 \leq \varepsilon$ – квадрат нормы антиградиента в точке должен стать меньше заданной точности.

Для первой функции:

Заданная точность	Начальная точка x_0	Точка минимума	Значение минимума	Количество итераций
0.1	(10, 10)	(-0.00425262, 2.97415998)	0.00070388	4
0.01	(10, 10)	(-0.00052142, 3.01055398)	0.00011193	3
0.00001	(10, 10)	(0.00000084, 3.00000080)	0.00000000	3

Таким образом, количество итераций стало ещё меньше, т. е. повысилась оптимальность поиска минимума.

Траектория алгоритма:

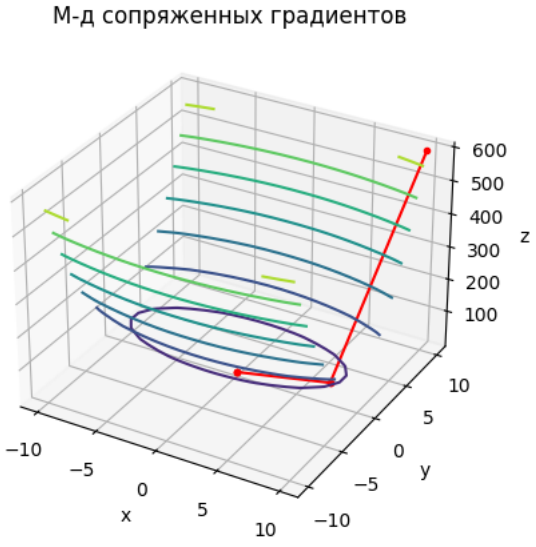


Для второй функции:

Заданная точность	Начальная точка x_0	Точка минимума	Значение минимума	Количество итераций
0.1	(10, 10)	(0.09839993, 0.01394507)	1.01065487	5
0.01	(10, 10)	(0.00024223, -0.00022658)	1.00000032	6
0.00001	(10, 10)	(0.00008497, -0.00000091)	1.00000001	4

Количество итераций при наших вводных данных уменьшается при росте точности, вероятно, из-за высокой погрешности поиска минимума при использовании метода золотого сечения – шаги получаются слишком большими.

Траектория алгоритма:



Из графика видно, что после первого шага, который не отличается от шагов в предыдущем методе, траектория схождения более уверенно ведёт к минимуму целевой функции.

Реализация метода:

```
def conjugate_gradient_descent(e, x0, func, part_deriv_x, part_deriv_y, antigrad_func):
    print("Метод сопряженных градиентов с e = {:0.8f}, начало в {}".format(e, x0))
    ax = plt.figure().add_subplot(projection='3d')
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("z")
    x_show, y_show = np.meshgrid(np.linspace(-10, 10, 20), np.linspace(-10, 10, 20))
    z_show = func(x_show, y_show)
    ax.contour(x_show, y_show, z_show)
    iterations = 1  # количество итераций
    x = x0
    fx0 = func(x[0], x[1])
    ag = antigrad(x[0], x[1], part_deriv_x, part_deriv_y)  # считаем антиградиент в текущей точке
(вектор)
    p = ag  # направление движения
    h = golden_cut(antigrad_func, x[0], x[1], p, e, 0, 100)  # вычисляем шаг
    x = [p[0] * h + x[0], p[1] * h + x[1]]  # делаем шаг в направлении убывания длиной h
    fx = func(x[0], x[1])
    ax.plot([x0[0], x[0]], [x0[1], x[1]], [fx0, fx], '.-', c='red')
    while True:
        iterations += 1
        if iterations > 1000000:
            print("Алгоритм не сошёлся!")
            break
        ag_prev = ag  # предыдущее значение антиградиента (вектор)
        p_prev = p  # предыдущее направление движения
        ag = antigrad(x[0], x[1], part_deriv_x, part_deriv_y)  # считаем антиградиент в текущей
точке (вектор)
        ag_norm = np.sqrt(ag[0] ** 2 + ag[1] ** 2)  # вычисляем норму (длину) вектора
антиградиента
        if ag_norm ** 2 < e:
            fx = func(x[0], x[1])  # вычисляем значение ф-и в новой точке
            print("Минимум в точке: [{:0.8f}, {:0.8f}], значение минимума: {:0.8f}, всего
итераций: {}"
                  .format(x[0], x[1], fx, iterations))
            break
        # применяем формулу Флетчера-Ривса для вычисления коэффицента для получения сопряженного
направления
        w = (ag[0] * ag[0] + ag[1] * ag[1]) / (ag_prev[0] * ag_prev[0] + ag_prev[1] * ag_prev[1])
        # получаем направление движения, сопряженное с направлением на предыдущем шаге
        p = [ag[0] + w * ag_prev[0], ag[1] + w * ag_prev[1]]
        h = golden_cut(antigrad_func, x[0], x[1], p, e, 0, 100)  # вычисляем длину шага
        x_prev = x
        fx_prev = fx
        x = [p[0] * h + x[0], p[1] * h + x[1]]  # делаем шаг в направлении движения длиной h
        fx = func(x[0], x[1])
        ax.plot([x_prev[0], x[0]], [x_prev[1], x[1]], [fx_prev, fx], '.-', c='red')
    plt.title("М-д сопряженных градиентов")
    plt.show()
```

Выводы:

В ходе лабораторной работы были изучены методы поиска минимума целевой функции, такие как метод градиентного спуска с постоянным и переменным шагом, метод наискорейшего спуска и метод сопряженных градиентов. Самым оптимальным для изучаемых функций оказался последний метод, обеспечивающий наименьшее количество вычислений.

Метод спуска с постоянным шагом применим только для тех случаев, когда начальная точка находится достаточно близко к искомой точке минимума. В противном случае, очень быстро растущее количество итераций может превысить установленные реализацией лимиты.

Метод с переменным шагом с использованием условия Армихо отличается от остальных методов параметром c, который, вероятно, зависим от изучаемой целевой функции. И удачный или неудачный его выбор может повлиять на результаты.
Методы наискорейшего спуска и сопряженных градиентов используют метод золотого сечения для минимизации вспомогательной функции. В нашей реализации мы использовали поиск на отрезке от 0 до 100 для возможных значений шага. Такая установка, возможно, будет не оптимальной для случаев, когда начальная точка находится далеко от искомого минимума, и получаемый шаг будет слишком маленьким.