Министр науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ

Национальный исследовательский университет ИТМО

Мегафакультет трансляционных информационных технологий

Факультет информационных технологий и программирования

Лабораторная работа № 3

По дисциплине «Прикладная математика»

Методы минимизация двумерной функции

Выполнили: Бонет Станислав,
Гусев Андрей,
Величко Максим
Группа М32061

Санкт-Петербург, 2023

Градиент - вектор, своим направлением указывающий направление наибольшего возрастания некоторой скалярной величины, значение которой меняется от одной точки пространства к другой, образуя скалярное поле, а по величине (модулю) равный скорости роста этой величины в этом направлении.

Градиентный спуск – метод нахождения локального минимума или максимума функции с помощью движения вдоль градиента. Для минимизации функции в направлении градиента используются методы одномерной оптимизации.

Основная идея метода заключается в том, чтобы идти в направлении наискорейшего спуска, а это направление задаётся антиградиентом $-\nabla f$:

$$x^{[k+1]} = x^{[k]} - \lambda^{[k]}\nabla f(x^{[k]}),$$

где $\nabla f = \frac{\partial f}{\partial x} i + \frac{\partial f}{\partial y} j + \frac{\partial f}{\partial z} l$,

$\lambda^{[k]}$ выбирается:

- постоянной, в этом случае метод может расходиться;
- дробным шагом, т.е. длина шага в процессе спуска делится на некое число;
- наискорейшим спуском: $\lambda^{[k]} = argmin\, f(x^{[k]} - \lambda\nabla f(x^{[k]}))$.

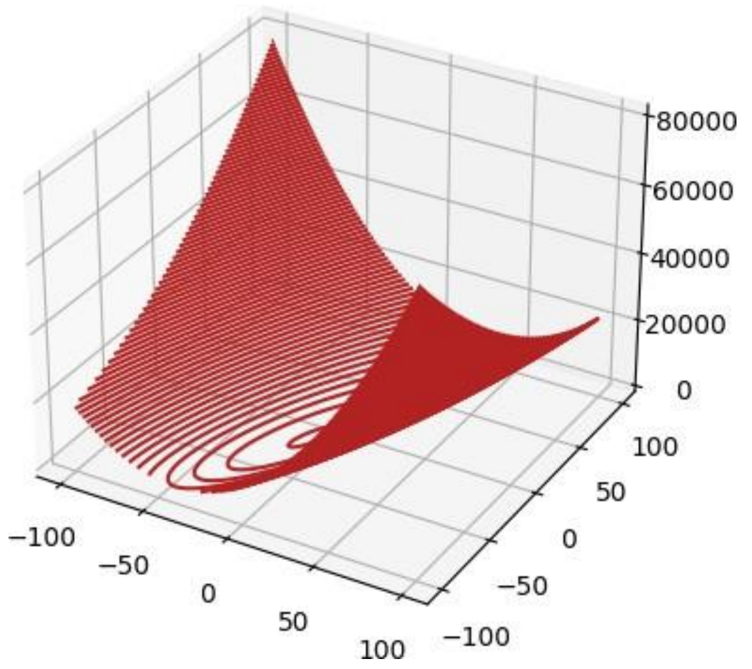
Алгоритм метода градиентного спуска:

1. Задаются ε и $x^{[k]}$ при $k = 0$;
2. Рассчитываются $x^{[k+1]} = x^{[k]} - \lambda^{[k]}\nabla f(x^{[k]})$;
3. Проверяется условие остановки:
   - Если $|x^{[k+1]} - x^{[k]}| > \varepsilon$, $|f(x^{[k+1]}) - f(x^{[k]})| > \varepsilon$ или $||\nabla f(x^{[k+1]})|| > \varepsilon$, то $k = k + 1$ и переход к шагу 2;
   - Иначе $x = x^{[k+1]}$ и остановка.

Исходная функция 1:

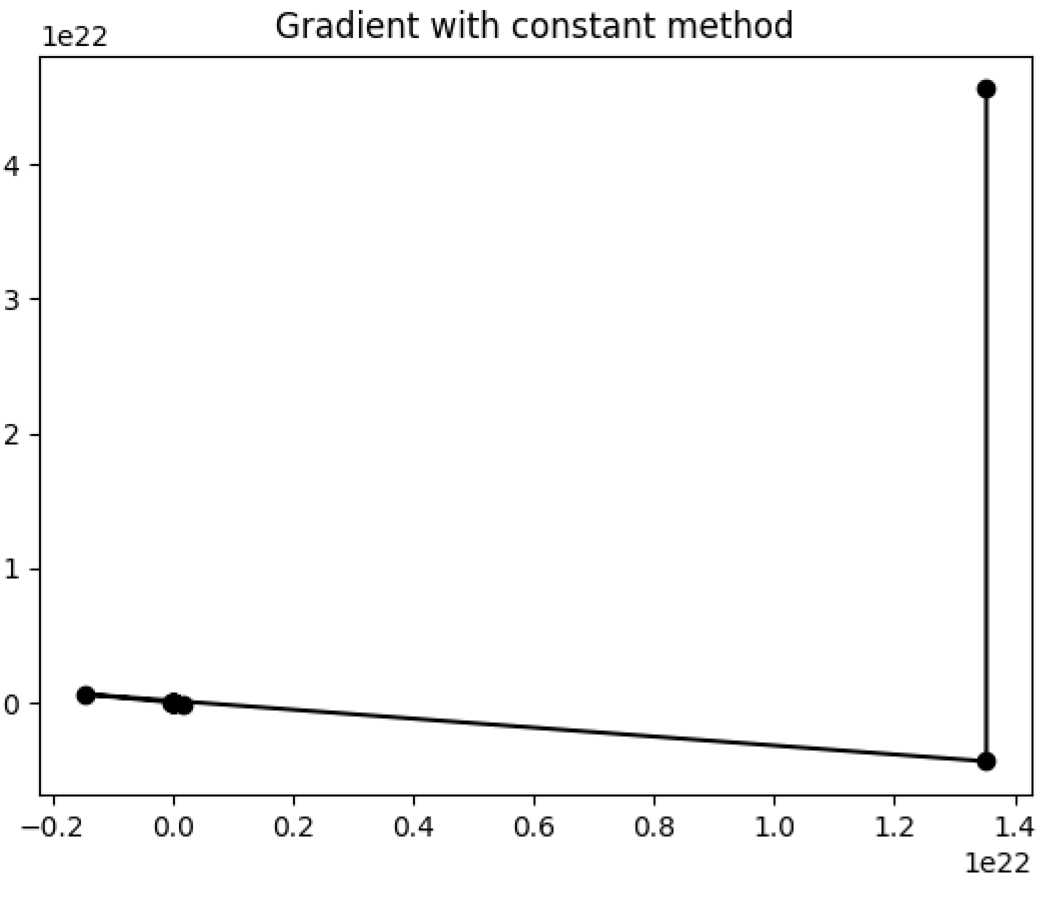
$f(x, y) = (2x + 5)^2 + (3 + y)^2 - 3xy$

$\varepsilon = 0,0001$



Метод с постоянной величиной шага:

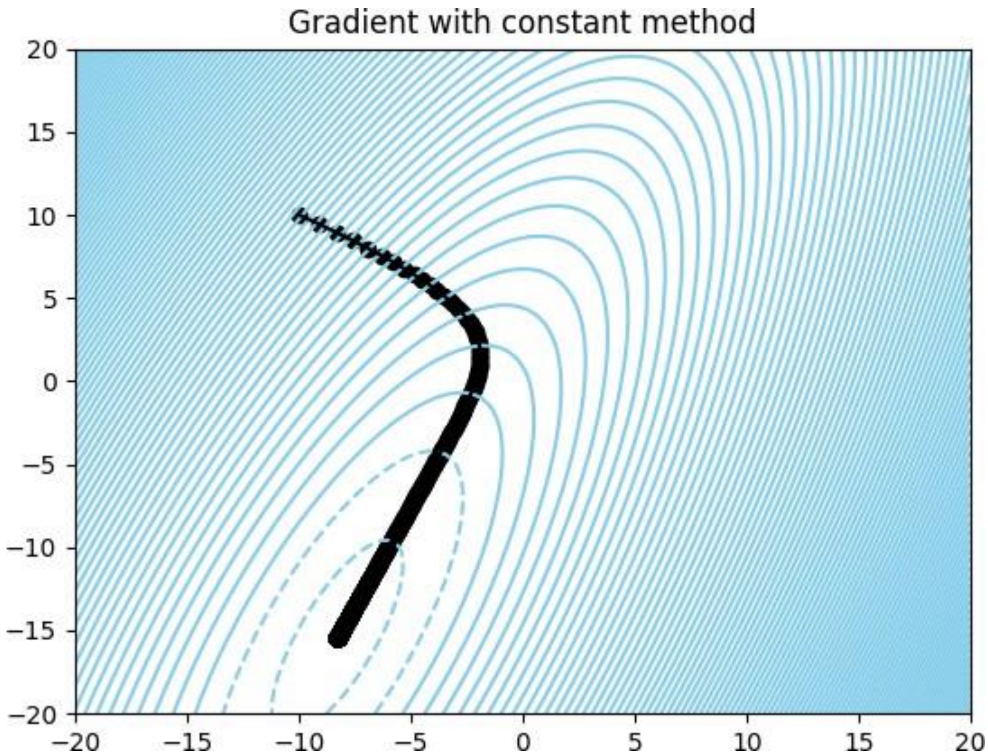
Рассматриваемая точка (-10; 10)



Количество итераций: 25

Шаг: 1

Минимум не найден

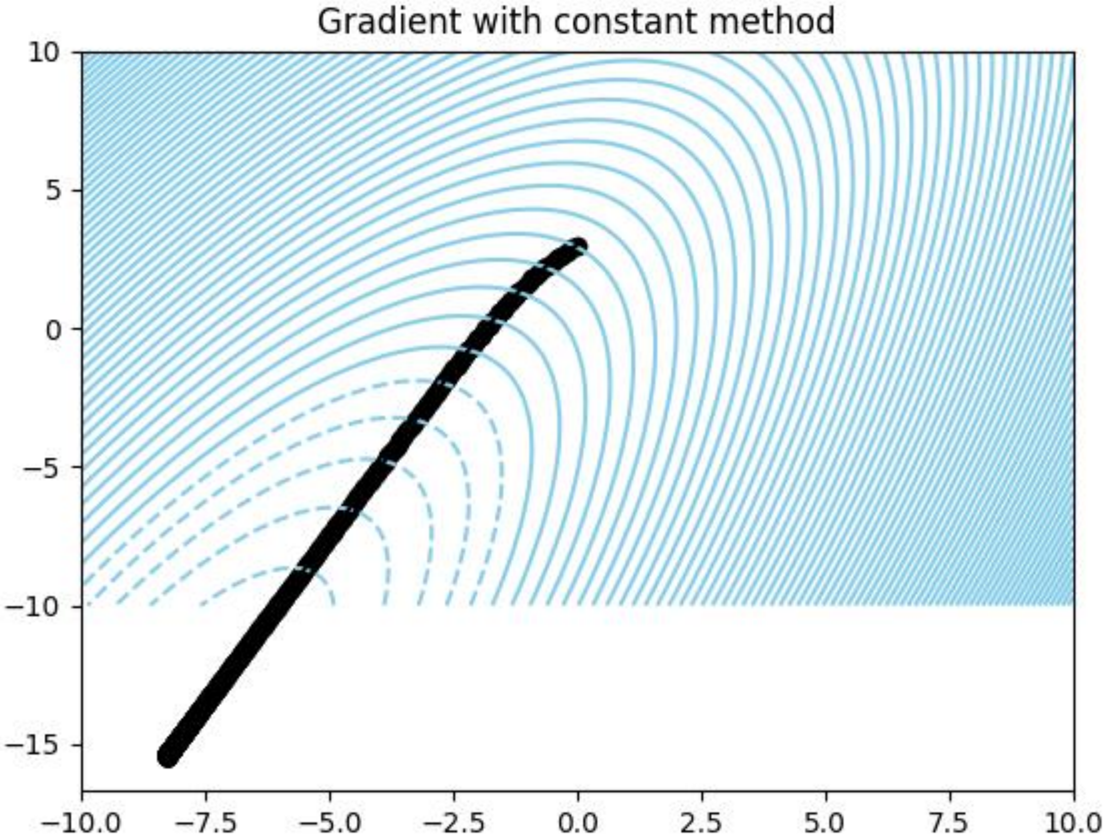


Количество итераций: 972

Шаг: 0,01

Найденный минимум: x = -8.2802750540505560868, y = -15.4272578655637429

Рассматриваемая точка (0; 3)



Количество итераций: 952

Шаг: 0,01

Найденный минимум: x = -8.280281717074470400728541311, y = - -15.415456047682663683804874 97

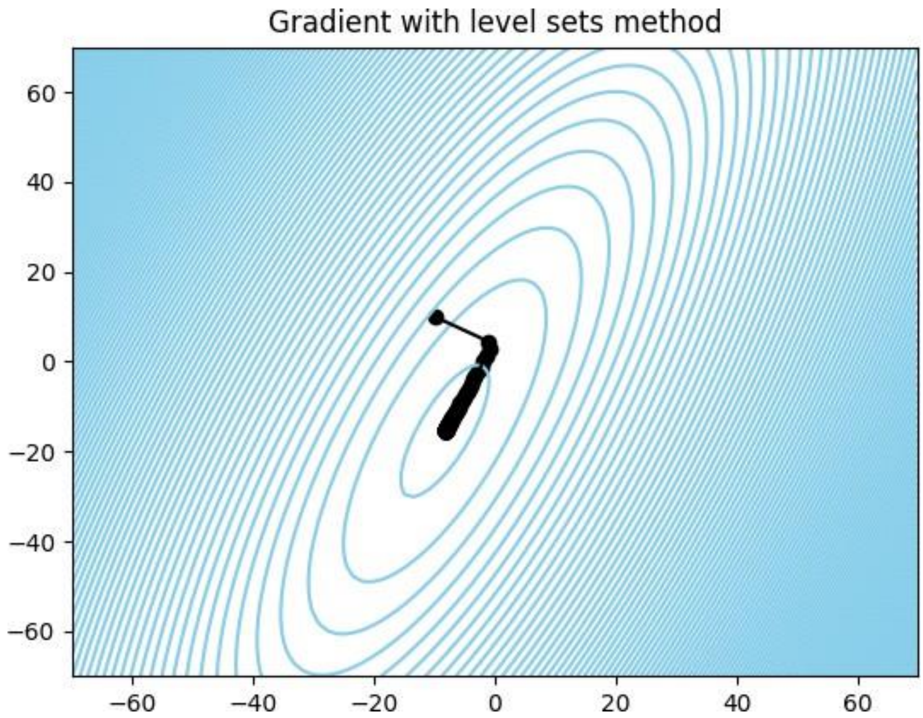
Код программы:

```
def gradient_descent_constant_step(function, gradient_function, initial_point,
learning_rate, num_iterations):
    """
    Градиентный спуск с постоянным шагом.

    Аргументы:
    - function: функция, для которой выполняется градиентный спуск
    - gradient_function: функция для вычисления градиента функции
    - initial_point: начальная точка (x0, y0) для градиентного спуска
    - learning_rate: постоянный шаг градиентного спуска
    - num_iterations: количество итераций градиентного спуска

    Возвращает:
    - optimal_point: оптимальная точка, достигнутая после градиентного спуска
    - optimal_value: оптимальное значение функции, достигнутое после
градиентного спуска
```

```
    """

    point = np.array(initial_point)  # Преобразуем начальную точку в numpy
массив

    for iteration in range(num_iterations):
        gradient = np.array(gradient_function(*point))  # Вычисляем градиент
функции в текущей точке

        new_point = point - learning_rate * gradient  # Вычисляем новую точку с
заданным шагом
        value = function(*new_point)  # Вычисляем значение функции в новой точке

        point = new_point  # Обновляем текущую точку

    return point, value


# Задание начальной точки, постоянного шага градиентного спуска и количества
итераций
initial_point = [0, 0]
learning_rate = 0.1
num_iterations = 100

# Применение градиентного спуска с постоянным шагом к первой функции
optimal_point1, optimal_value1 = gradient_descent_constant_step(f2, f2_gradient,
initial_point, learning_rate, num_iterations)
print("Optimal Point for f2:", optimal_point1)
```

Метод дробления шага:

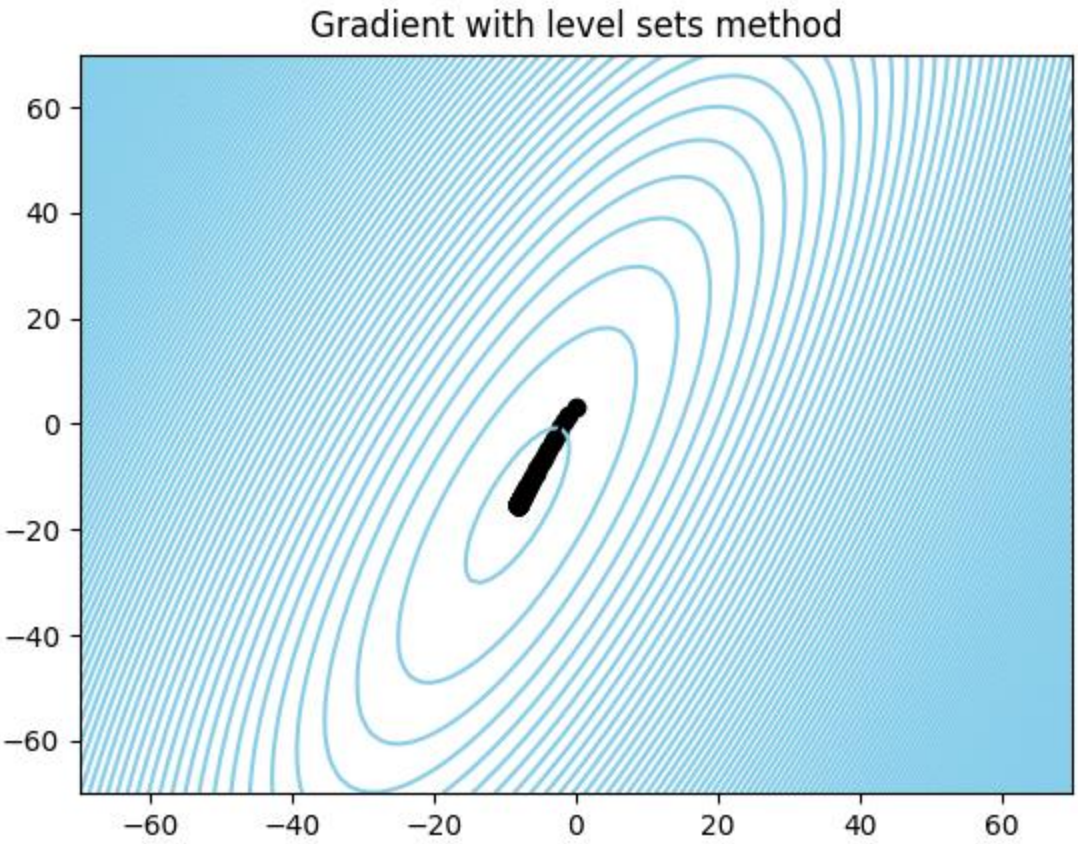
Рассматриваемая точка (-10; 10)



Количество итераций: 124

Найденный минимум: x = -8.28517179343448308 51, y =
-15.42844836909704509 23

Рассматриваемая точка (0; 3)



Количество итераций: 123

Найденный минимум: x = -8.285195201114702706781850530, y = -15.427318247491096267263194 68

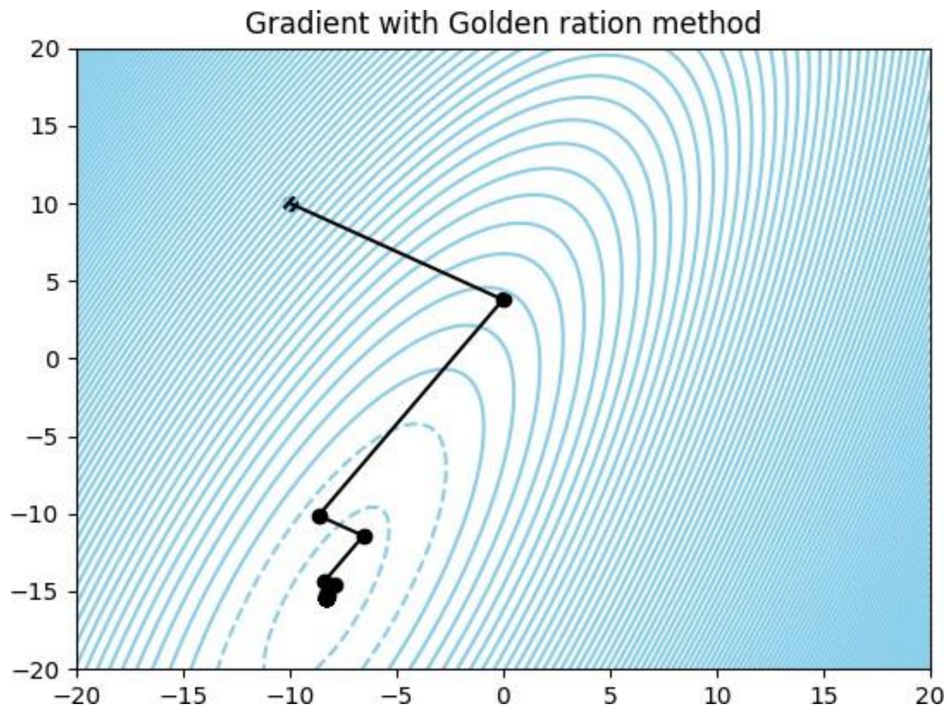
Код программы:

```
def armijo_condition(function, point, gradient, step_size, c, tau):
    """
    Проверка условия Армихо для выбора подходящего шага градиентного спуска.

    Аргументы:
    - function: функция, для которой выполняется градиентный спуск
    - point: текущая точка (x, y) для проверки условия
    - gradient: градиент функции в текущей точке
    - step_size: текущий шаг градиентного спуска
    - c: параметр условия Армихо (0 < c < 1)
    - tau: коэффициент дробления шага (0 < tau < 1)

    Возвращает:
    - True, если условие Армихо выполняется
    - False, если условие Армихо не выполняется
    """

    new_point = point - step_size * gradient  # Вычисляем новую точку с заданным
шагом
    function_value = function(*point)  # Значение функции в текущей точке
    new_function_value = function(*new_point)  # Значение функции в новой точке
```

```
    return new_function_value <= function_value - c * step_size *
np.linalg.norm(gradient) ** 2


def gradient_descent_with_armijo(function, gradient_function, initial_point,
initial_step_size, c, tau, max_iterations):
    """
    Метод градиентного спуска с дроблением шага и использованием условия Армихо.

    Аргументы:
    - function: функция, для которой выполняется градиентный спуск
    - initial_point: начальная точка (x0, y0) для градиентного спуска
    - initial_step_size: начальный шаг градиентного спуска
    - c: параметр условия Армихо (0 < c < 1)
    - tau: коэффициент дробления шага (0 < tau < 1)
    - max_iterations: максимальное количество итераций

    Возвращает:
    - optimal_point: оптимальная точка, достигнутая после градиентного спуска
    - optimal_value: оптимальное значение функции, достигнутое после
градиентного спуска
    """

    def gradient(x, y):
        return np.array(gradient_function(x, y))

    point = np.array(initial_point)  # Преобразуем начальную точку в numpy
массив
    step_size = initial_step_size  # Инициализируем шаг градиентного спуска
    optimal_point = point  # Инициализируем оптимальную точку значением
начальной точки
    optimal_value = function(*point)  # Вычисляем значение функции в начальной
точке
    iteration = 0  # Счет

    while iteration < max_iterations:
        gradient_value = gradient(*point)  # Вычисляем градиент функции в
текущей точке

        if armijo_condition(function, point, gradient_value, step_size, c, tau):
            new_point = point - step_size * gradient_value  # Вычисляем новую
точку с заданным шагом
            value = function(*new_point)  # Вычисляем значение функции в новой
точке

            if value < optimal_value:
                optimal_point = new_point  # Обновляем оптимальную точку, если
значение функции уменьшилось
                optimal_value = value

            point = new_point  # Обновляем текущую точку
            iteration += 1  # Увеличиваем счетчик итераций
```

```
        else:
            step_size *= tau  # Дробим шаг градиентного спуска

    return optimal_point, optimal_value

# Задание начальной точки, начального шага градиентного спуска, параметров
Армихо и максимального количества итераций
initial_point = [0, 0]
initial_step_size = 0.1
c = 0.5
tau = 0.5
max_iterations = 100

# Применение градиентного спуска с дроблением шага и условием Армихо к первой
функции
optimal_point1, optimal_value1 = gradient_descent_with_armijo(f1, f1_gradient,
initial_point, initial_step_size, c, tau, max_iterations)
print("Optimal Point for f1:", optimal_point1)
```

Метод золотого сечения:



Количество итераций: 17

Найденный минимум: x = -8.285720668456103236, y = -15.428568947581858302 8

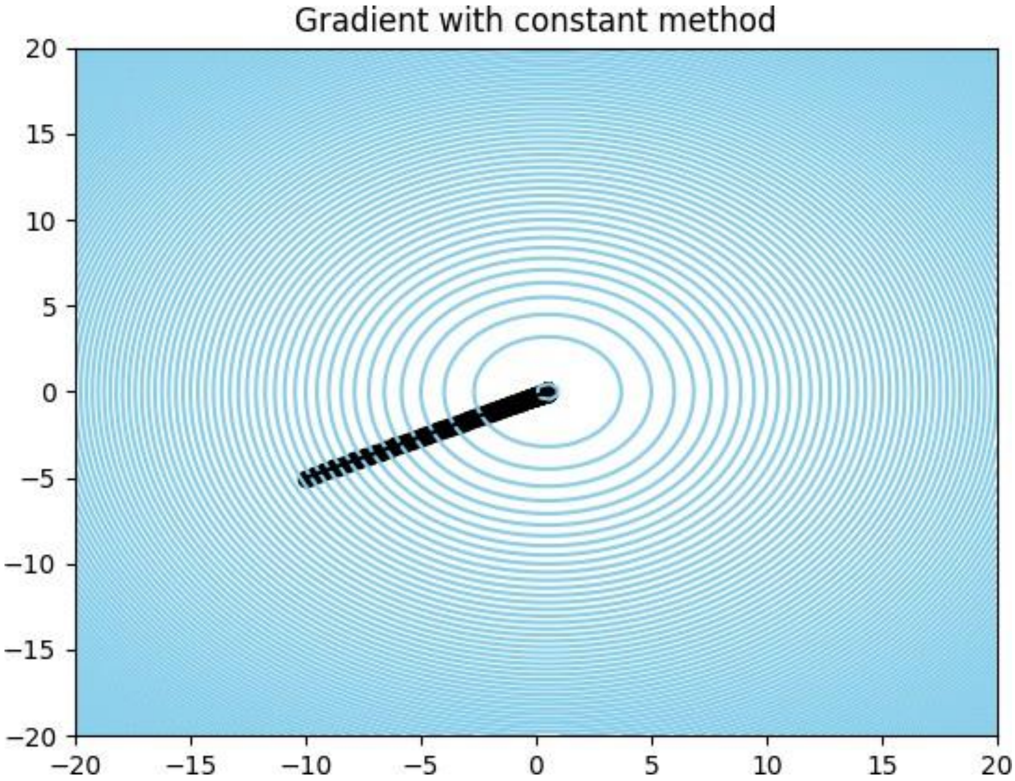
Код программы:

```
ef line_search_golden_section(function, point, direction, initial_step_size,
epsilon):
    """
    Метод золотого сечения для одномерной оптимизации.

    Аргументы:
    - function: функция, для которой выполняется одномерная оптимизация
    - point: текущая точка (x, y) для одномерной оптимизации
    - direction: направление для одномерной оптимизации
    - initial_step_size: начальный шаг для одномерной оптимизации
    - epsilon: точность для определения условия остановки

    Возвращает:
    - step_size: оптимальный шаг, найденный методом золотого сечения
    """

    a = 0
    b = initial_step_size
    tau = (np.sqrt(5) - 1) / 2  # Значение золотого сечения

    # Функция для вычисления значения функции в заданной точке
    def f(alpha):
        return function(*(point + alpha * direction))

    # Начальные значения
    alpha1 = b - tau * (b - a)
    alpha2 = a + tau * (b - a)
```

```python
    f1 = f(alpha1)
    f2 = f(alpha2)

    while b - a > epsilon:
        if f1 < f2:
            b = alpha2
            alpha2 = alpha1
            alpha1 = b - tau * (b - a)
            f2 = f1
            f1 = f(alpha1)
        else:
            a = alpha1
            alpha1 = alpha2
            alpha2 = a + tau * (b - a)
            f1 = f2
            f2 = f(alpha2)

    step_size = (a + b) / 2  # Оптимальный шаг

    return step_siz
```

Метод с постоянной величиной шага:

Начальная точка (-10; -5)

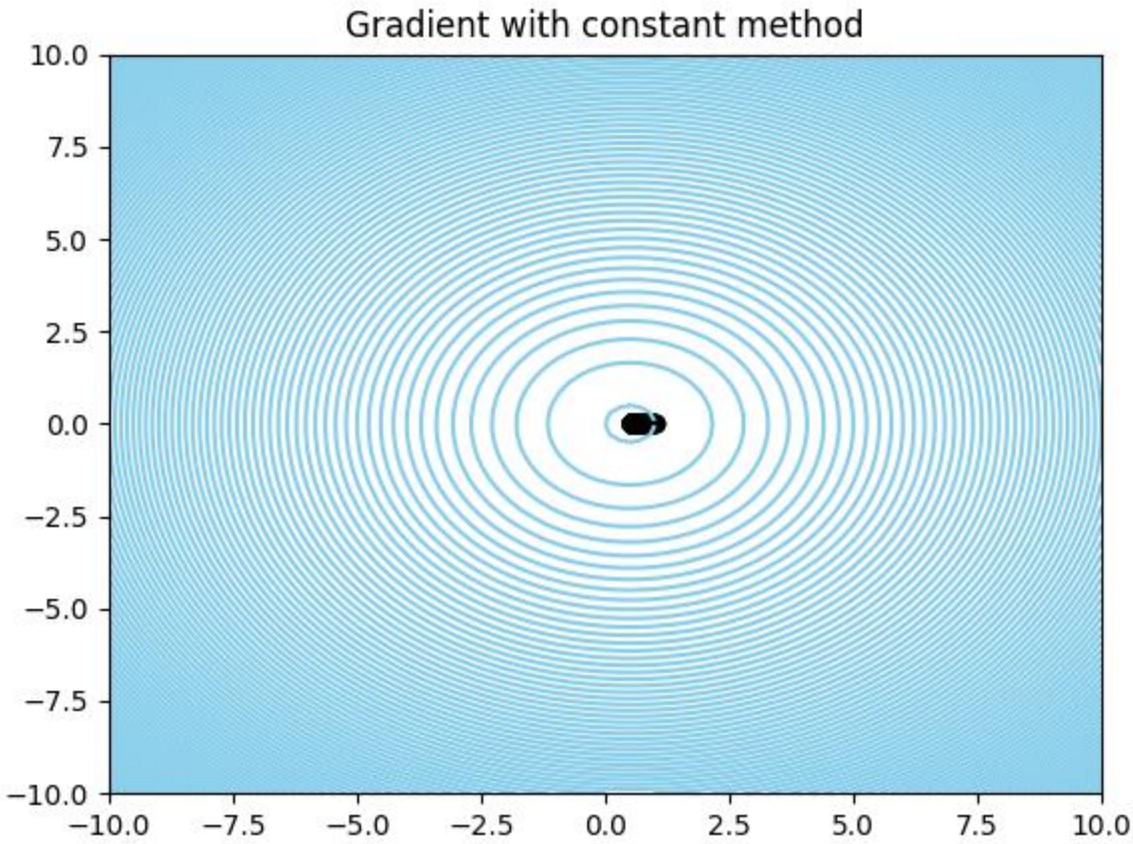


Количество итераций: 494

Шаг: 0,01

Найденный минимум: x = 0.4995038138753775797, y = -0.00023627910696305729

Начальная точка (1;0)

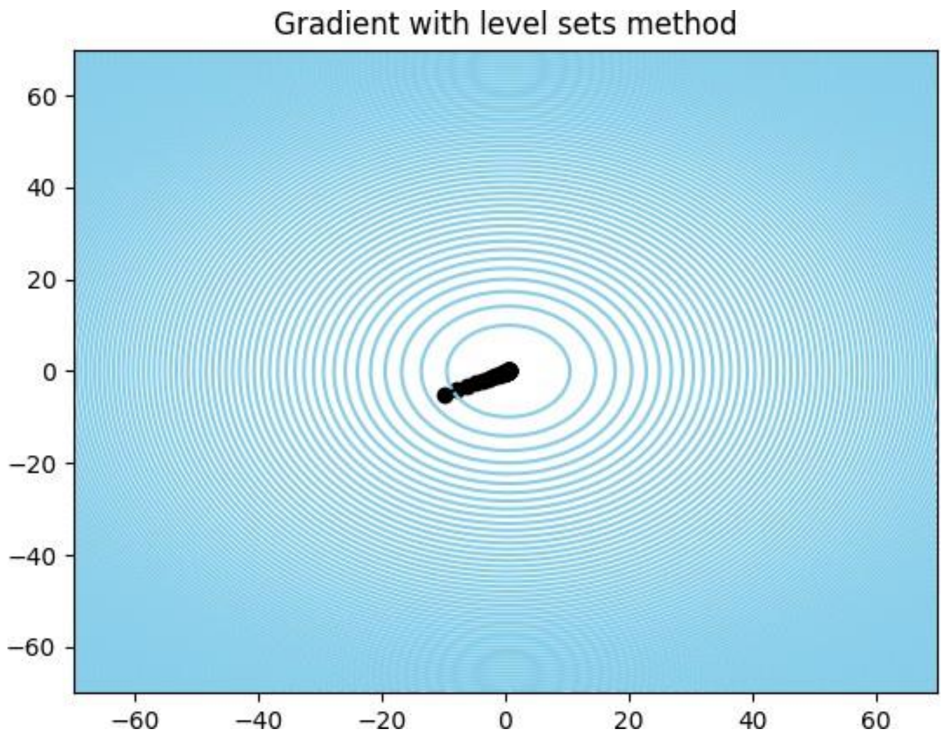


Количество итераций: 229

Шаг: 0.01

Найденный минимум: x = 0.5049947673279657815216071916, y = 0E-54

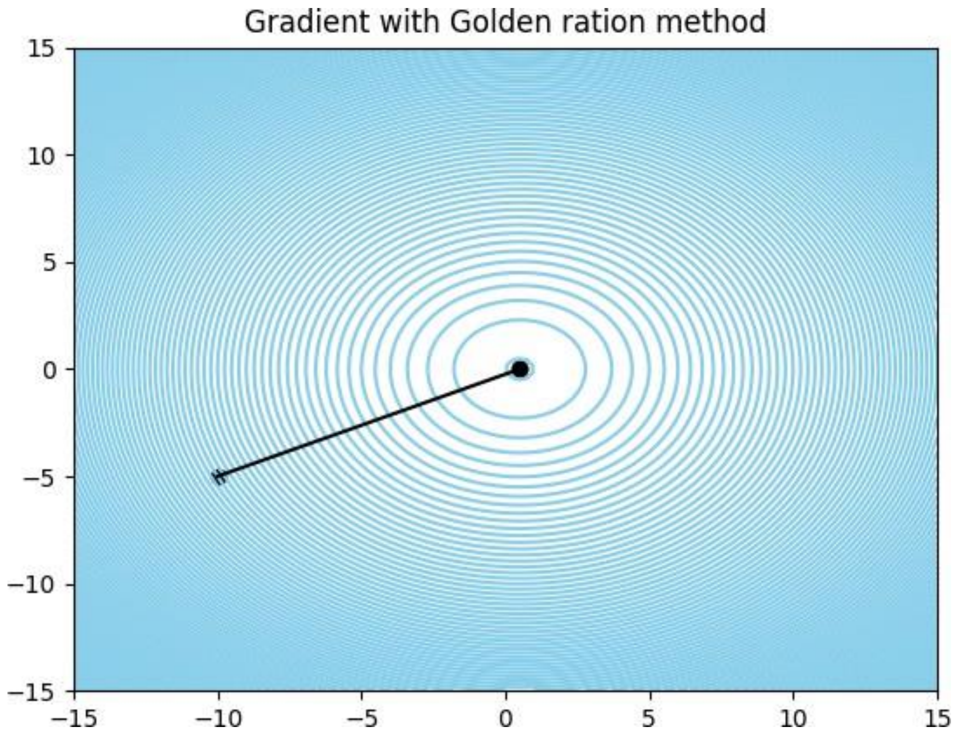
Метод дробления шага:



Количество итераций: 56

Найденный минимум: x = 0.49995089354498 5682, y =
-0.000023384026197 2944288

Метод золотого сечения:



Количество итераций: 3

Найденный минимум: x = 0.49999999998853, y = -0,0000546041426847

Получается, что градиентный спуск с методом золотого сечения и Фибоначчи позволяет достичь максимальной точности значительно быстрее в отличие от градиентного спуска с постоянным шагом, который может расходиться при неправильном выборе шага, градиентный спуск с дроблением шага требует внимательного подбора параметра α, который определяет, каким образом шаг будет дробиться. Однако правильный выбор этого параметра позволяет методу работать достаточно быстро.

Метод сопряженных градиентов

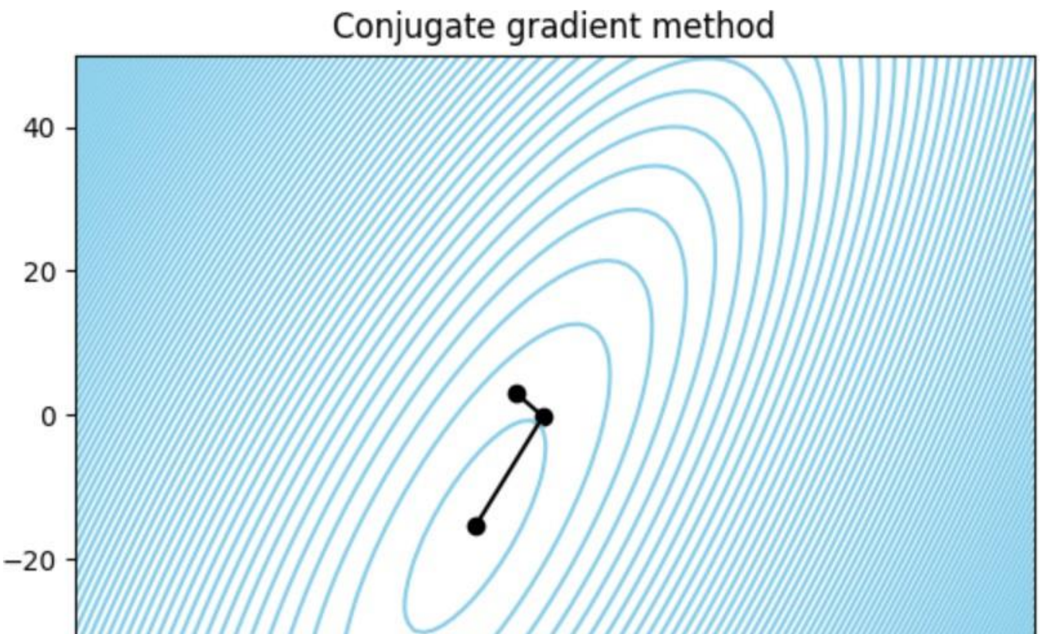
Метод сопряженных градиентов - метод нахождения локального экстремума функции на основе информации о её значениях и её градиенте. В случае квадратичной функции в R^n минимум находится не более чем за n шагов.

Теперь сравним траектории, полученные методом градиентного спуска с разными величинами шага с методом сопряженных градиентов для тех же функций:

1) $f(x, y) = (2x + 5)^2 + (3 + y)^2 - 3xy$

$\varepsilon = 0,001$

Начальная точка (-4; 3)



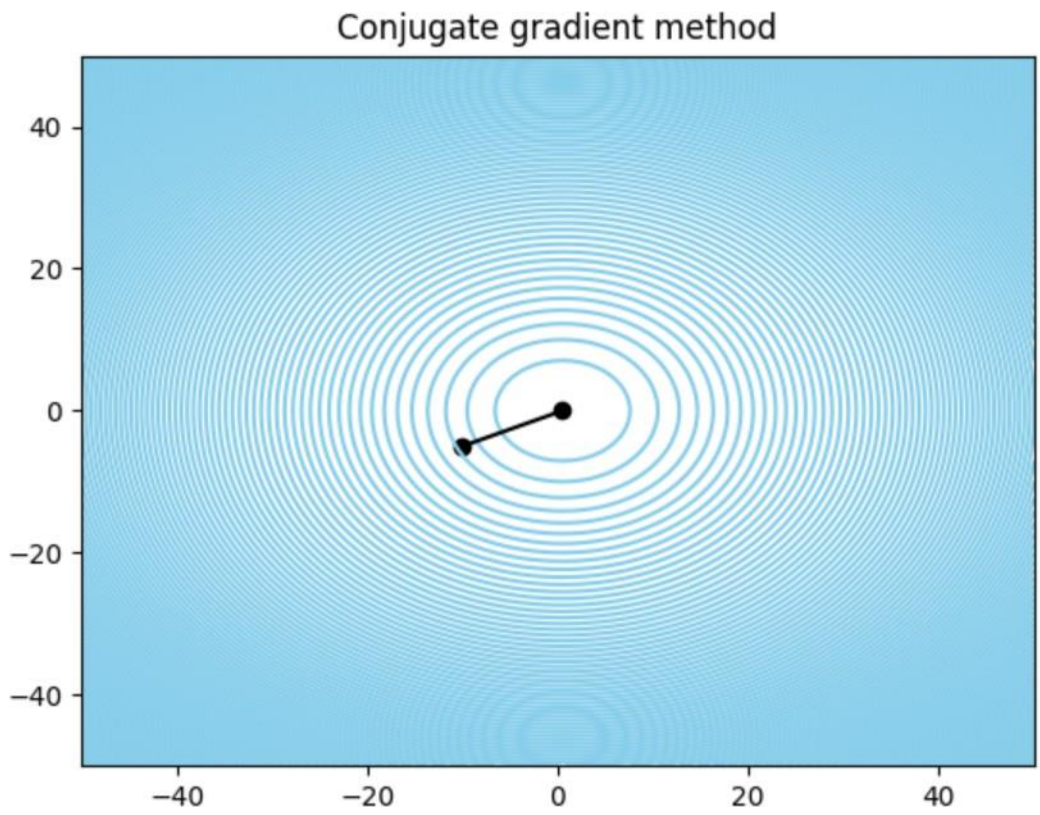
Количество итераций: 2

Найденный минимум: x = -8.285714285714285, y = -15.428571428571423

2) $f(x, y) = x^2 + y^2 - x$

$\varepsilon = 0,001$

Начальная точка (-10; -5)



Количество итераций: 1

Найденный минимум: $x = 0.5$, $y = 0.0$

Код программы:

```
def conjugate_gradient_method(function, gradient_function, hessian_function,
initial_point, epsilon, max_iterations):
    """
    Метод сопряженных градиентов для оптимизации квадратичной функции.

    Аргументы:
    - function: функция, для которой выполняется оптимизация
    - initial_point: начальная точка (x0, y0) для оптимизации
    - epsilon: точность для определения условия остановки
    - max_iterations: максимальное количество итераций

    Возвращает:
    - optimal_point: оптимальная точка, достигнутая после оптимизации
    - optimal_value: оптимальное значение функции, достигнутое после оптимизации
    """

    point = np.array(initial_point, dtype=np.float64)  # Преобразуем начальную
точку в numpy массив
    gradient = np.array(gradient_function(*point))  # Вычисляем градиент функции
в начальной точке
    direction = -gradient  # Направление спуска
    iteration = 0  # Счетчик итераций

    while iteration < max_iterations and np.linalg.norm(gradient) > epsilon:
```

```
        alpha = -(np.dot(gradient, direction) / np.dot(direction,
hessian_function(*point)))  # Шаг
        point += alpha * direction  # Вычисляем градиент функции в новой точке

        new_gradient = np.array(gradient_function(*point))  # Вычисляем новую
точку с заданным шагом
        beta = np.dot(new_gradient, new_gradient) / np.dot(gradient,
gradient)  # Коэффициент

        direction = -new_gradient + beta * direction  # Обновляем направление
спуска
        gradient = new_gradient
        iteration += 1  # Вычисляем значение функции в новой точке

    return point, function(*point)


# Задание начальной точки, точности и максимального количества итераций
initial_point = [0, 0]
epsilon = 1e-3
max_iterations = 1000

# Применение метода сопряженных градиентов к первой функции
optimal_point1, optimal_value1 = conjugate_gradient_method(f1, f1_gradient,
f1_hessian, initial_point, epsilon, max_iterations)
print("Optimal Point for f1:", optimal_point1)
```

Таким образом, метод сопряженных градиентов работает быстрее и эффективнее, чем метод градиентного спуска с различными величинами шага. Мы на примере смогли убедиться, что количество итераций не превышает размерность пространства, в котором мы работаем, при этом у нас получилось подобрать функцию, поиск минимума на которой у нас отрабатывает за одну итерацию.

Также мы реализовали генератор случайных квадратичных функций:

```
def generate_quadratic_function(n, k):
    A = np.random.rand(n, n)  # Генерируем случайную матрицу размером (n, n)
    A = (A + A.T) / 2  # Делаем матрицу симметричной
    eigvals = np.linalg.eigvalsh(A)  # Вычисляем собственные значения матрицы

    # Масштабируем собственные значения для получения заданного числа
обусловленности
    scaled_eigvals = (k / np.max(eigvals)) * eigvals

    # Создаем квадратичную функцию с матрицей A и собственными значениями
scaled_eigvals
    def quadratic_function(x):
        return 0.5 * np.dot(x.T, np.dot(A, x))

    return quadratic_function
```

```python
# Пример использования
n = 3  # Количество переменных
k = 10  # Число обусловленности
quadratic_func = generate_quadratic_function(n, k)

# Генерируем случайную точку
x = np.random.rand(n)

# Вычисляем значение квадратичной функции в точке x
result = quadratic_func(x)
print("Result:", result)
```