Национальный исследовательский университет ИТМО
Факультет информационных технологий и программирования
Прикладная математика и информатика

Методы спуска

Отчёт по лабораторной работе №3

Работу выполнили:
Гуревич Михаил
Трохан Александр

Преподаватель:
Москаленко Мария Александровна

Санкт-Петербург
2023

Лабораторная работа №3

Выполнили: Гуревич Михаил и Трохан Александр, M32001

✅ Полный код лабораторной работы с комментариями можно найти на **Github**, а отчёт в **Notion**.

Цель работы

Реализация и анализ методов многомерного спуска на различных функциях.

Постановка задачи

1. Реализовать различные методы спуска.
2. Проанализировать траектории предложенных алгоритмов на примере квадратичных функций, используя две-три различные функции, на которых работа методов отличается.
3. Для каждой функции сравнить эффективность методов с точки зрения количества вычислений, исследовать работу методов в зависимости от выбора начальной точки и построить графики с траекториями методов.
4. Реализовать генератор случайных квадратичных функций.
5. Исследовать зависимость числа итераций $T(n, k)$, необходимых градиентному спуску для сходимости в зависимости от размера пространства $2 \leq n \leq 10^3$ и числа обусловленности оптимизируемой функции $1 \leq k \leq 10^3$.

Ход работы

Реализация алгоритмов

Для начала реализуем класс квадратичной функции, где будет функция вычисления градиента:

```
class QuadraticFunction:
    def __init__(self, a, b, c=0):
        self.a = a
        self.b = b
        self.c = c

    @staticmethod
    def from_coefficients(x2, y2, xy, x, y, c):
        # xy / 2 - чтобы не учитывать дважды при сложении
        a = np.array([[x2, xy / 2], [xy / 2, y2]])
        b = np.array([x, y])
        return QuadraticFunction(a, b, c)

    def coefficients(self):
        return self.a[0, 0], self.a[1, 1], self.a[0, 1] * 2, self.b[0], self.b[1], self.c

    def __call__(self, point: np.ndarray) -> np.number:
        # @ - оператор матричного умножения
        return point @ self.a @ point.T + self.b @ point.T + self.c
```

```
    def gradient(self, point: np.ndarray):
        # т.к. матрица A симметрична, то градиент можно вычислить следующим образом:
        return 2 * self.a @ point.T + self.b

    def dimensions(self):
        return self.a.shape[0]

    def __str__(self):
        if self.dimensions() == 2:
            return f"{self.a[0, 0]}x^2 + {self.a[0, 1] * 2}xy + {self.a[1, 1]}y^2 + {self.b[0]}x + {self.b[1]}y + {self.c}"
        return f"A: {self.a}\nB: {self.b}\nC: {self.c}"
```

Теперь реализуемый следующие методы многомерной оптимизации:

Метод градиентного спуска с постоянным шагом

Описание метода: осуществляется оптимизация по направлению антиградиента функции $-\nabla f$, так как это является направлением наискорейшего убывания функции в конкретной точке. Метод стартует из наперёд заданной точки, $\alpha = \text{const}$ — заранее выбранный постоянный шаг, а каждая следующая точка вычисляется по формуле:

$$x_{k+1} = x_k - \alpha \nabla f(x_k);\ k = 0, 1, 2...$$

Тогда если выполняется условие $f(x_{k+1}) < f(x_k)$, то последовательность $\{x_k\}$ сходится к минимуму x^*.

Реализация метода:

```
def gradiend_descent(f: QuadraticFunction, x_0, step, epsilon, max_iterations=2e6):
    x_values = [x_0]
    f_values = [f(x_0)]
    for _ in range(int(max_iterations)):
        x = x_values[-1]
        grad = f.gradient(x)
        if np.linalg.norm(grad) < epsilon:
            break
        x_values.append(x - step * grad)
        f_values.append(f(x_values[-1]))
    else:
        print("Maximal number of iterations exceeded")

    return x_values, f_values
```

Метод градиентного спуска с дроблением шага по условию Армихо

Описание метода: будем вычислять точки последовательности аналогично предыдущему методу, однако вместо постоянного шага будем проверять выполнение условия:

$$f(x_k) - f(x_k - \alpha_k \nabla f(x_k)) \geq \epsilon \cdot \alpha_k ||\nabla f(x_k)||^2$$

При этом начальный шаг α_0 и коэффициент ϵ заданы наперёд. В случае невыполнения данного условия коэффициент α_k уменьшается (например, в 2 раза) до тех пор, пока условие не будет выполняться. Такое изменение шага гарантировано обеспечивает выполнение условия $f(x_{k+1}) < f(x_k)$ и сходимость последовательности $\{x_k\}$.

Реализация метода: в лабораторной работе был реализован метод с делением коэффициента α_k пополам и $\epsilon = 0.5$.

```
def gradiend_descent_armijo(f: QuadraticFunction, x_0, step, epsilon, max_iterations=2e6, armijo_coef=0.5):
    x_values = [x_0]
    f_values = [f(x_0)]
    for _ in range(int(max_iterations)):
        x = x_values[-1]
        grad = f.gradient(x)
        if np.linalg.norm(grad) < epsilon:
            break
        # уменьшаем шаг, пока не выполнится условие Армихо:
        # f(x) - f(x - step * grad) >= armijo_coef * step * np.linalg.norm(grad) ** 2
        while f(x) - f(x - step * grad) < armijo_coef * step * np.linalg.norm(grad) ** 2:
            step /= 2
        x_values.append(x - step * grad)
        f_values.append(f(x_values[-1]))
    else:
        print("Maximal number of iterations exceeded")

    return x_values, f_values
```

Метод наискорейшего градиентного спуска

Описание метода: в данном методе градиентного спуска шаг выбирается по следующей формуле:

$$\alpha_k = \underset{\alpha \in [0; +\infty)}{\arg\min} f(x_k - \alpha \nabla f(x_k))$$

Иначе говоря, выбирается направление убывания функции (то есть направление антиградиента в точке), а затем находится такой шаг, при котором в следующей точке будет минимум функции на этом луче. Оптимальный шаг можно вычислить с помощью методов одномерной оптимизации.

Реализация метода: в данном случае в качестве метода одномерной оптимизации используется метод золотого сечения, а также выбор оптимального шага происходит от 0 до 1, а не до $+\infty$.

```
def golden_section(f: FunctionType, left, right, epsilon):
    while right - left > epsilon:
        delta = (right - left) / PHI
        x_1 = right - delta
        x_2 = left + delta

        if f(x_1) < f(x_2):
            right = x_2
        else:
            left = x_1

    return (left + right) / 2

def steepest_gradiend_descent(f: QuadraticFunction, x_0, epsilon, max_iterations=2e6):
    x_values = [x_0]
    f_values = [f(x_0)]
    for _ in range(int(max_iterations)):
        x = x_values[-1]
        grad = f.gradient(x)
        if np.linalg.norm(grad) < epsilon:
            break
        # ищем оптимальный шаг методом золотого сечения
        step = golden_section(lambda step, x=x, grad=grad: f(x - step * grad), 0, 1, epsilon)
        x_values.append(x - step * grad)
        f_values.append(f(x_values[-1]))
    else:
        print("Maximal number of iterations exceeded")

    return x_values, f_values
```

Метод сопряжённых градиентов

Описание метода: суть метода заключается в том, что при выборе направления на $k+1$-ой итерации рассматривается ещё и градиент на k-ой итерации. Таким образом, новое направление можно будет вычислить как:

$$p_k = \nabla f(x_k) + \beta_k \nabla f(x_{k-1}), \beta_k = \frac{||\nabla f(x_k)||^2}{||\nabla f(x_{k-1})||^2}$$

Тогда:

$$x_{k+1} = x_k - \alpha_k \cdot p_k$$

При этом α_k можно выбирать с помощью методов одномерной оптимизации аналогично предыдущему алгоритму.

Реализация метода:

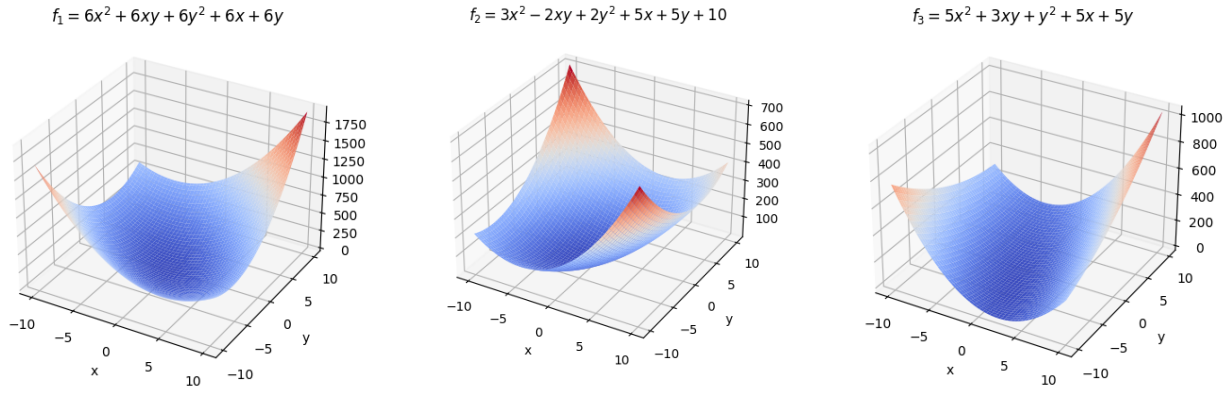
```
def conjugate_gradient(f: QuadraticFunction, x_0, epsilon, max_iterations=2e6):
    x_values = [x_0]
    f_values = [f(x_0)]
    for _ in range(int(max_iterations)):
        x = x_values[-1]
        grad = f.gradient(x)
        if np.linalg.norm(grad) < epsilon:
            break
        # вместо градиента берем направление, образуемое предыдущим и текущим градиентами
        if len(x_values) > 1:
            prev_grad = f.gradient(x_values[-2])
            beta = np.linalg.norm(grad) ** 2 / np.linalg.norm(prev_grad) ** 2
            grad = grad + beta * prev_grad
        # ищем оптимальный шаг методом золотого сечения
        step = golden_section(lambda step, x=x, grad=grad: f(x - step * grad), 0, 1, epsilon)
        x_values.append(x - step * grad)
        f_values.append(f(x_values[-1]))
    else:
        print("Maximal number of iterations exceeded")

    return x_values, f_values
```

Анализ работы методов

Выбор функций

Для начала выберем три квадратичные функции, на которых проверим работу наших методов. Функции и их графики представлены ниже:

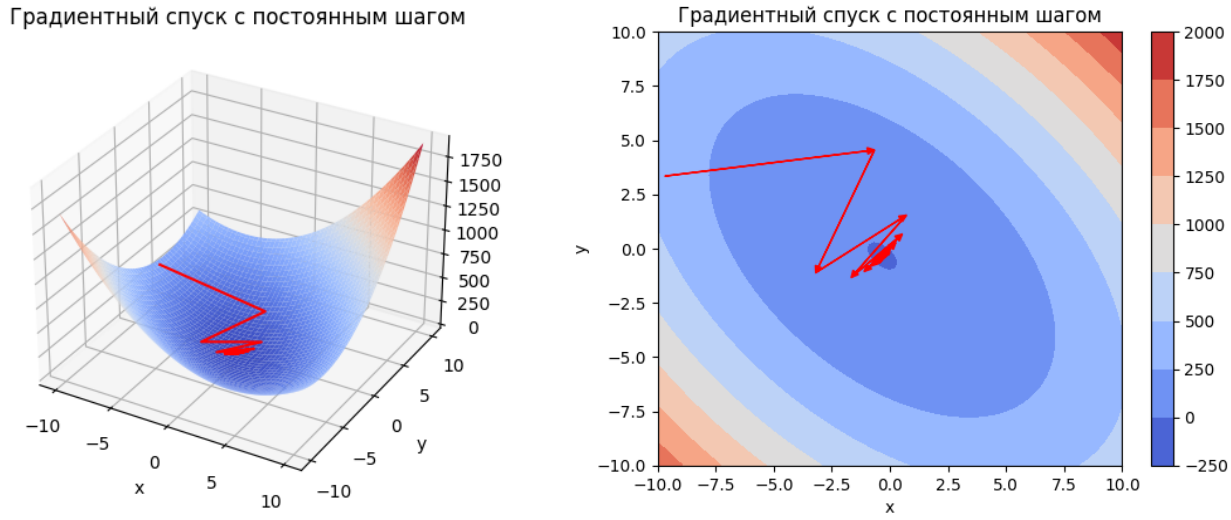


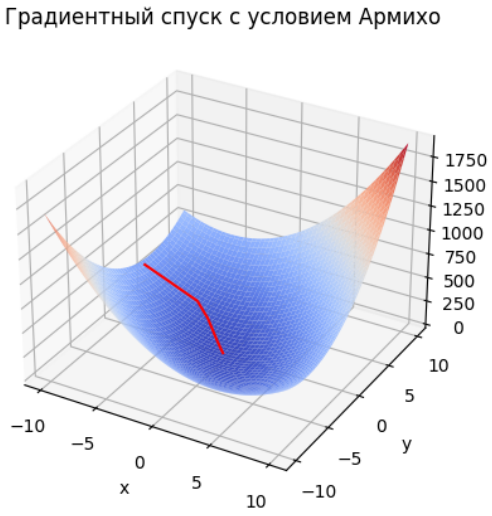
Траектории методов

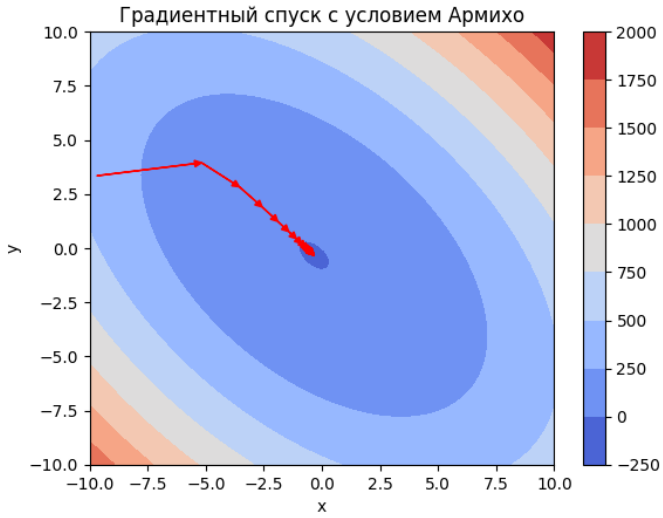
Рассмотрим то, между какими точками происходит перемещение при работе метода (то есть точки последовательности $\{x_k\}$). Для каждой функции будет представлено 8 графиков: по 2 на каждый метод, где первый — траектория метода в трёхмерном пространстве и второй — траектория метода на линиях уровня функции.

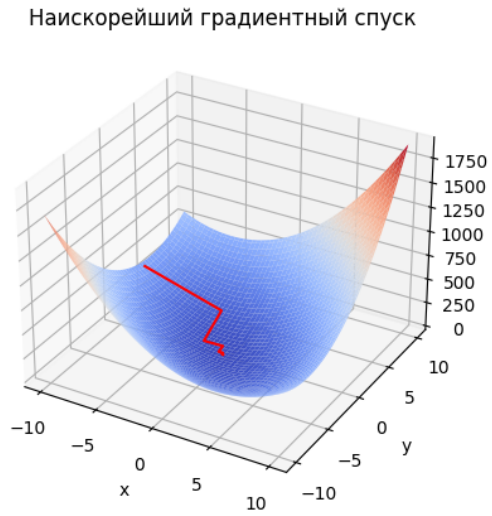
В качестве старта была выбрана случайная начальная точка $(-9.68842686; 3.34143661)$.

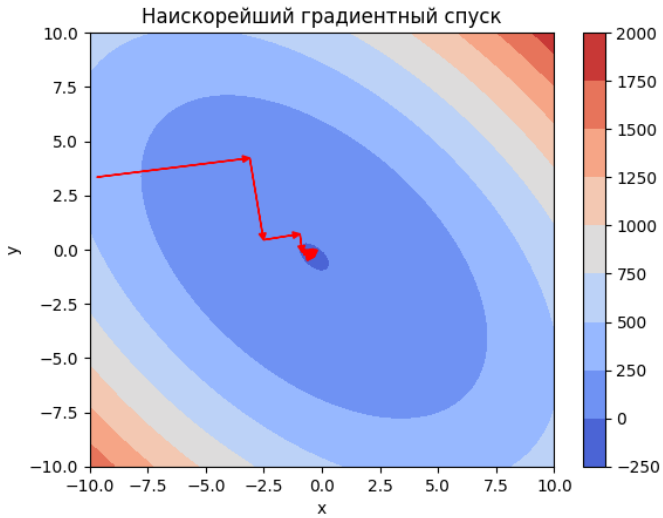
1. **Функция $f_1(x, y) = 6x^2 + 6xy + 6y^2 + 6x + 6y$:**

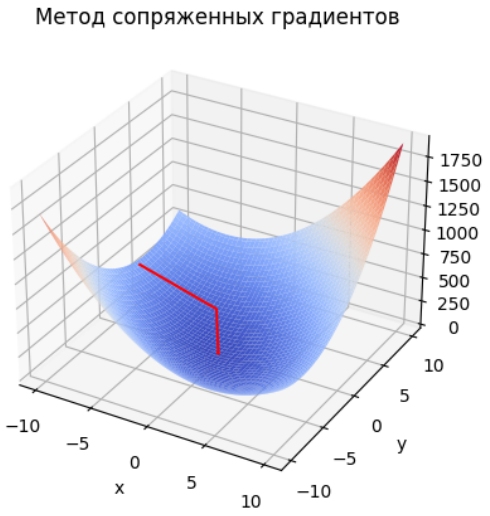


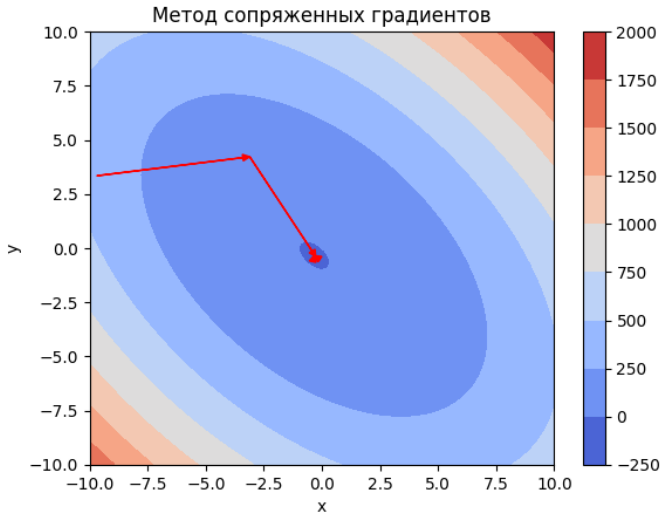

Градиентный спуск с условием Армихо
1750
1500
1250
1000
750
500
250
0
−10
−5
0
5
10
x
y


Градиентный спуск с условием Армихо
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
x
y
2000
1750
1500
1250
1000
750
500
250
0
−250

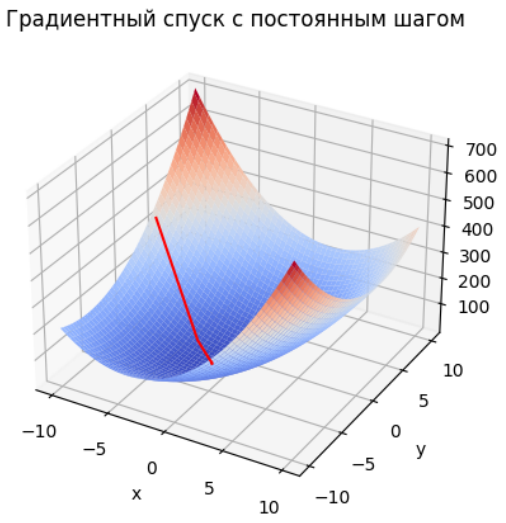

Наискорейший градиентный спуск
1750
1500
1250
1000
750
500
250
0
−10
−5
0
5
10
x
y

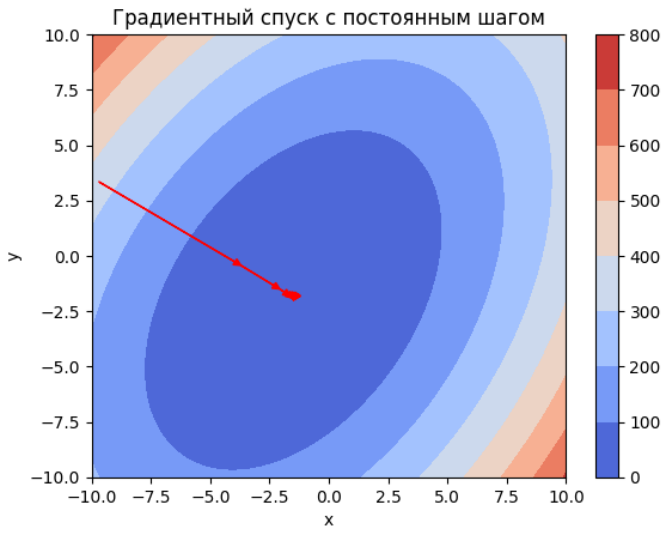

Наискорейший градиентный спуск
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
x
y
2000
1750
1500
1250
1000
750
500
250
0
−250

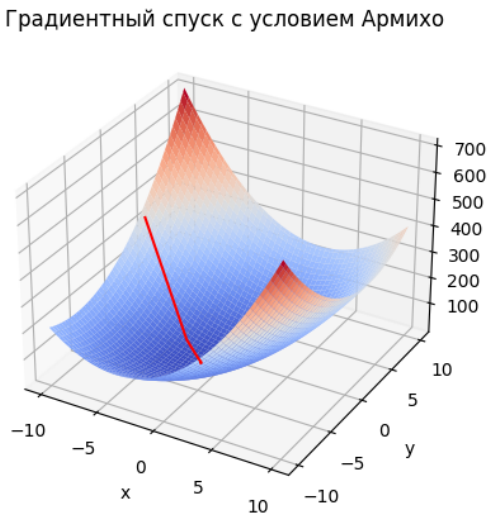


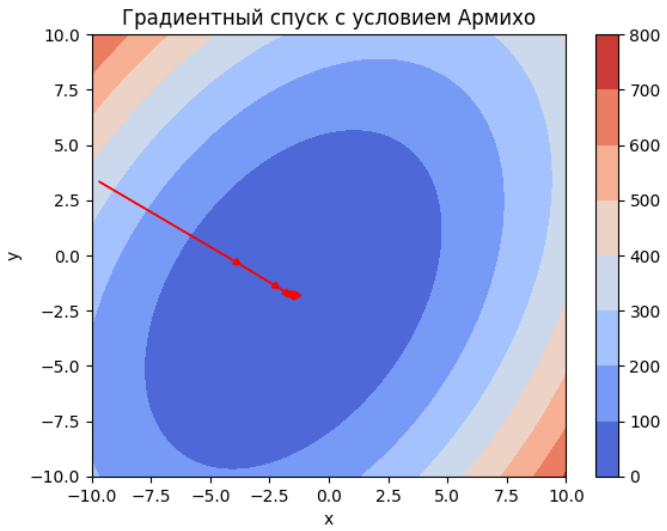


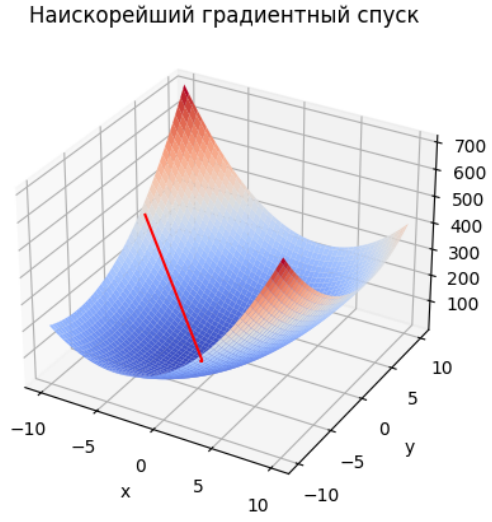
2. **Функция** $f_2(x, y) = 3x^2 - 2xy + 2y^2 + 5x + 5y + 10$:

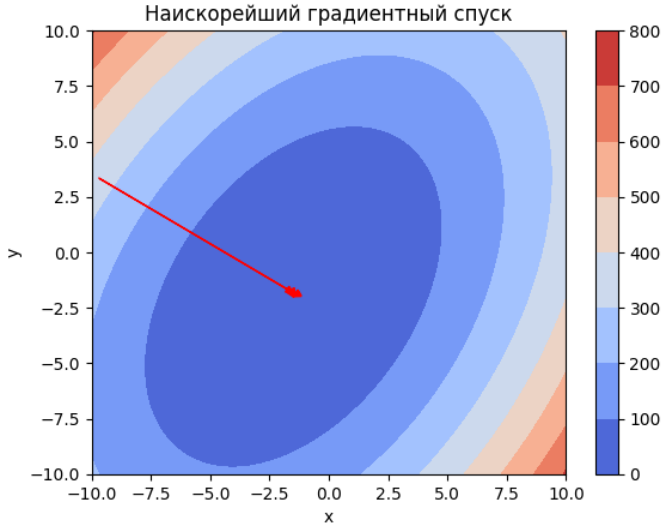


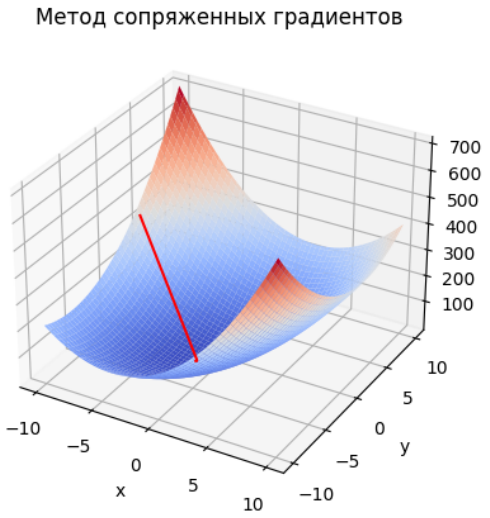


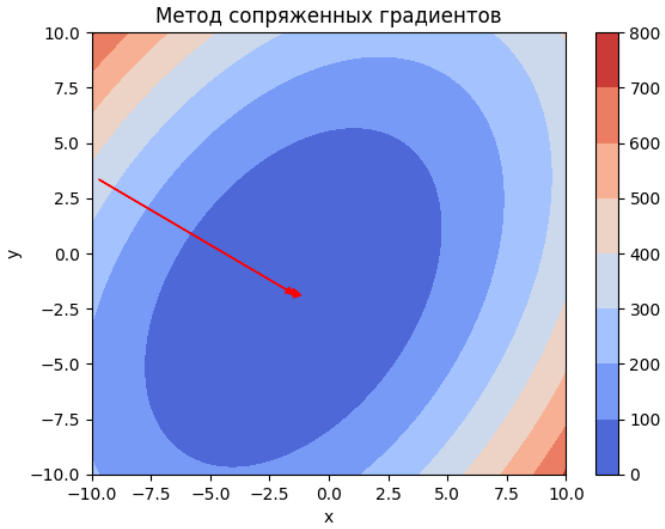

Градиентный спуск с условием Армихо
700
600
500
400
300
200
100
−10
−5
0
5
10
x
y


Градиентный спуск с условием Армихо
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
x
y
800
700
600
500
400
300
200
100
0

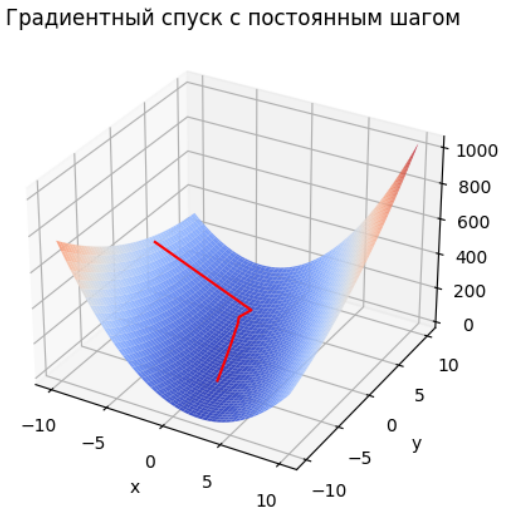

Наискорейший градиентный спуск
700
600
500
400
300
200
100
−10
−5
0
5
10
x
y

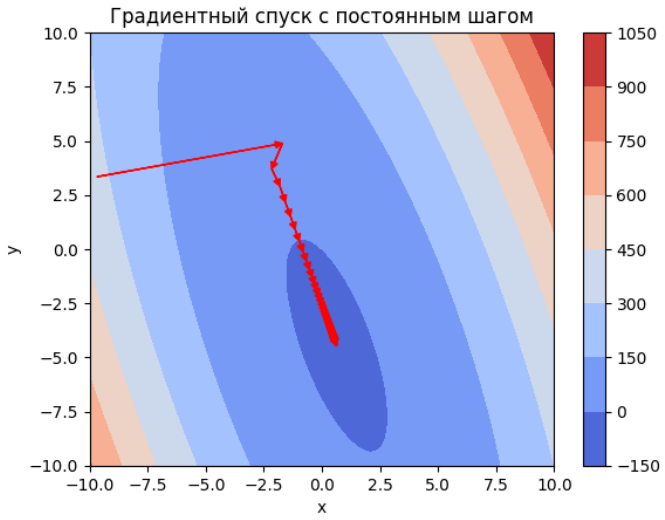

Наискорейший градиентный спуск
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
x
y
800
700
600
500
400
300
200
100
0

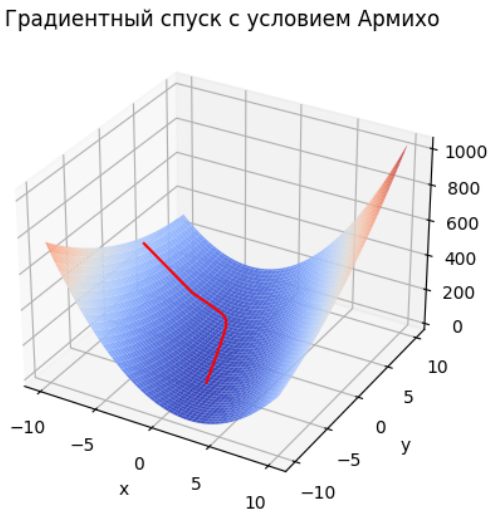


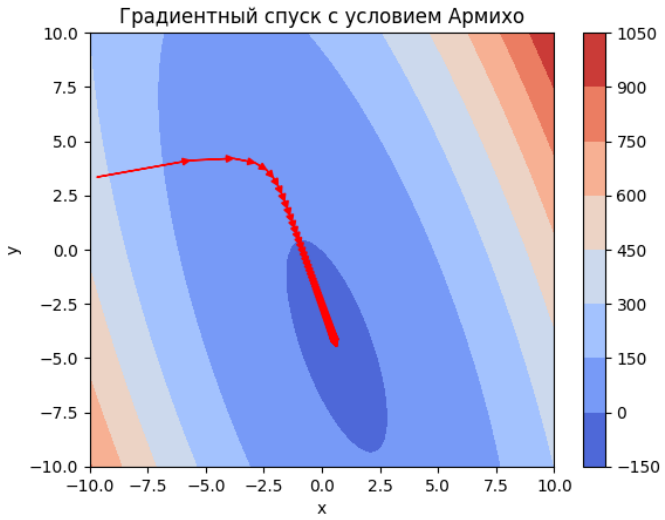


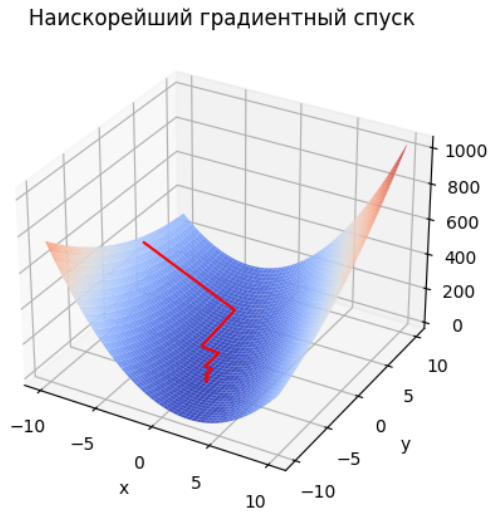
3. **Функция** $f_3(x, y) = 5x^2 + 3xy + y^2 + 7x + 7y$**:**

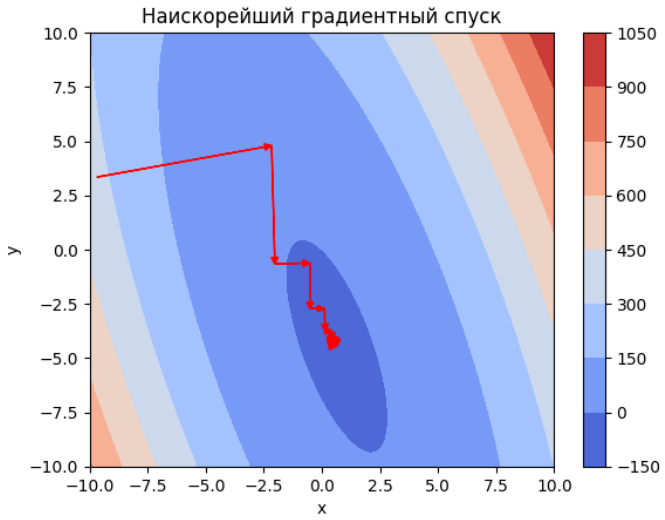


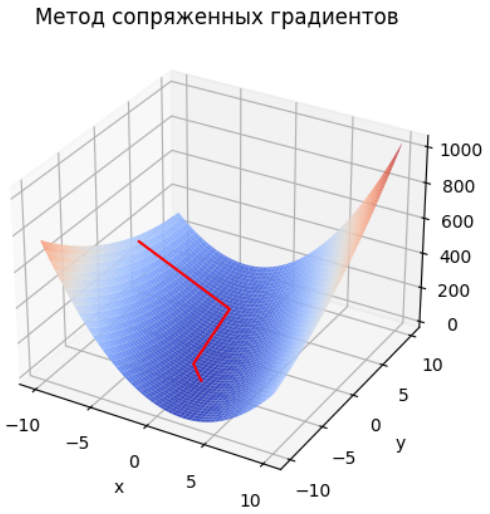


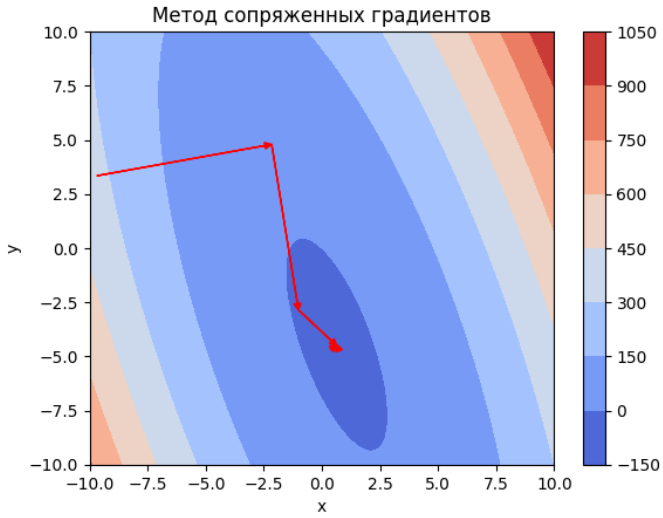

Градиентный спуск с условием Армихо
1000
800
600
400
200
0
−10
−5
0
5
10
x
y


Градиентный спуск с условием Армихо
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
x
y
1050
900
750
600
450
300
150
0
−150


Наискорейший градиентный спуск
1000
800
600
400
200
0
−10
−5
0
5
10
x
y


Наискорейший градиентный спуск
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
x
y
1050
900
750
600
450
300
150
0
−150





Выводы:

- При неудачном выборе начальной точки метод градиентного спуска с постоянным шагом будет иметь плохую траекторию сходимости, которая будет постоянно меняться и медленно сходиться к точке минимума.
- При удачном выборе начальной точки все методы будут иметь оптимальную траекторию, однако методы, в которых шаг подбирается на основе методов одномерной оптимизации сделают заметно меньше итераций.
- Метод сопряженных градиентов имеет самую лучшую траекторию для всех рассмотренных функций.

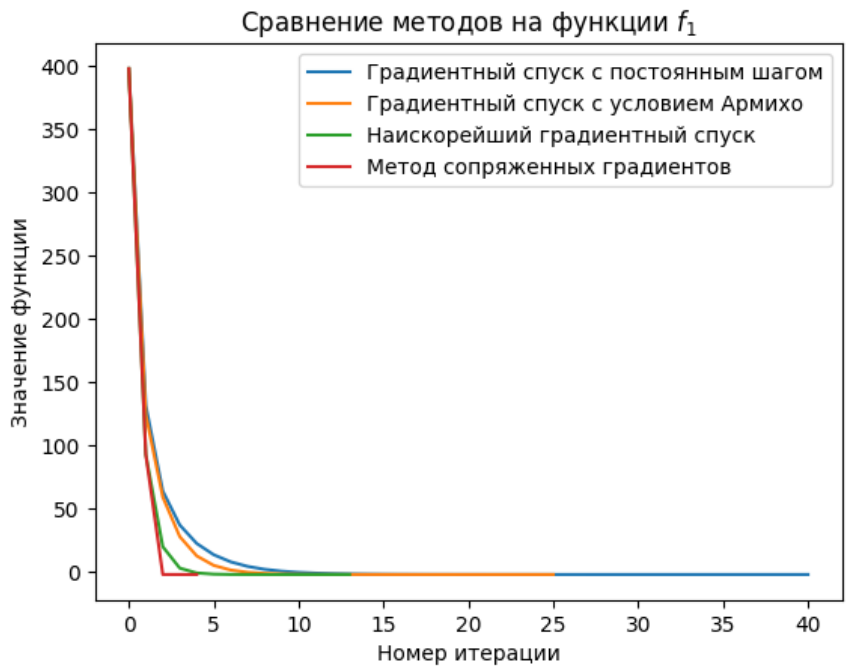
Сходимость метода градиентного спуска с постоянным шагом

Для всех рассмотренных функций метод градиентного спуска сошёлся, однако это может произойти не всегда. При неудачном выборе шага (обычно, если он слишком большой) метод может не сойтись, так как не будет выполняться условие $f(x_{k+1}) < f(x_k)$и последовательность $\{x_k\}$ разойдётся, то есть методу не хватит точности для того, чтобы попасть в точку минимума.

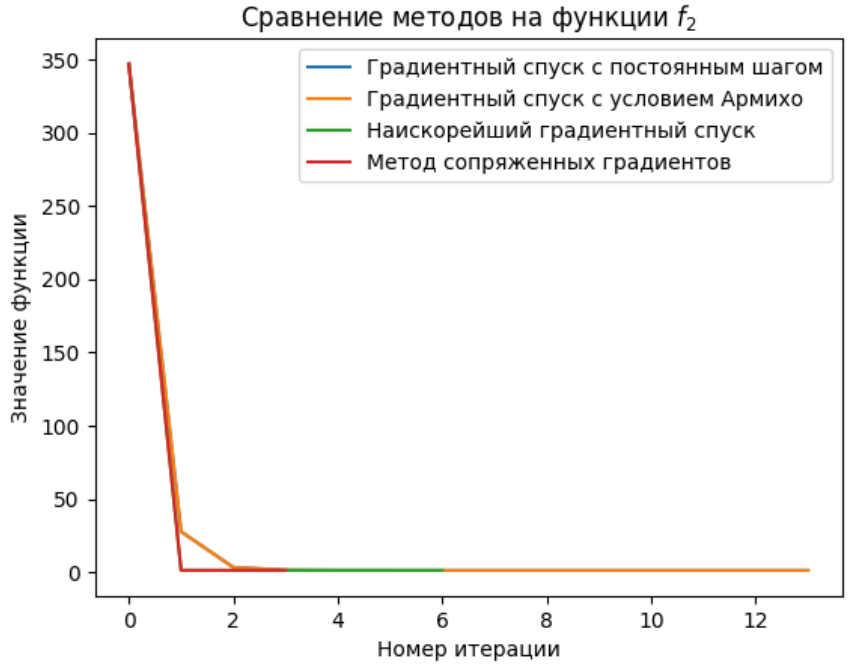
Количество итераций методов и количество вычислений градиентов

Для каждой функции рассмотрим число итераций, которое совершил каждый из методов. При этом число вычислений градиента прямо пропорционально числу итераций, и при должных оптимизациях можно достичь всего одного вычисления градиента на итерацию.

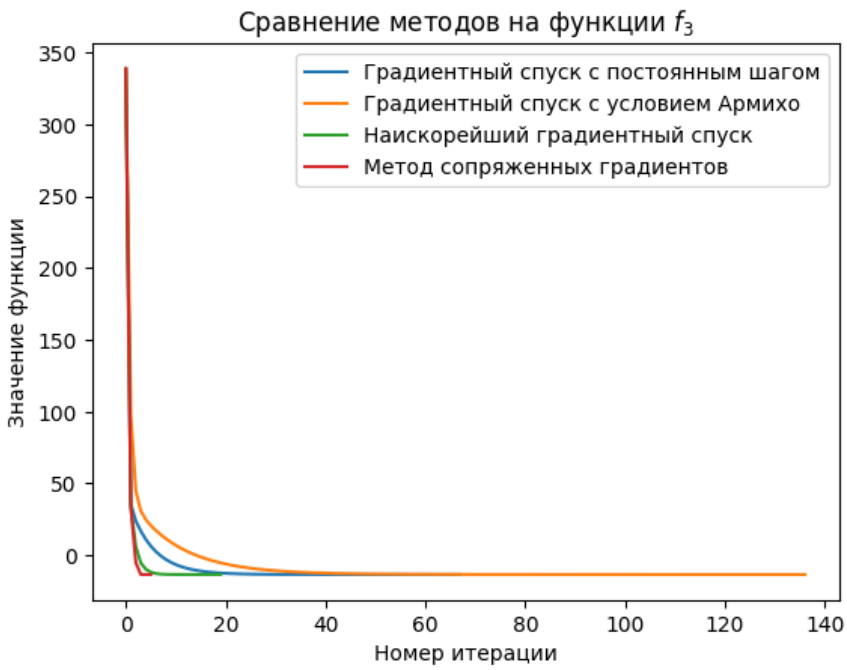
1. **Функция** $f_1(x, y) = 6x^2 + 6xy + 6y^2 + 6x + 6y$**:**



2. **Функция** $f_2(x, y) = 3x^2 - 2xy + 2y^2 + 5x + 5y + 10$:



3. **Функция** $f_3(x, y) = 5x^2 + 3xy + y^2 + 7x + 7y$:



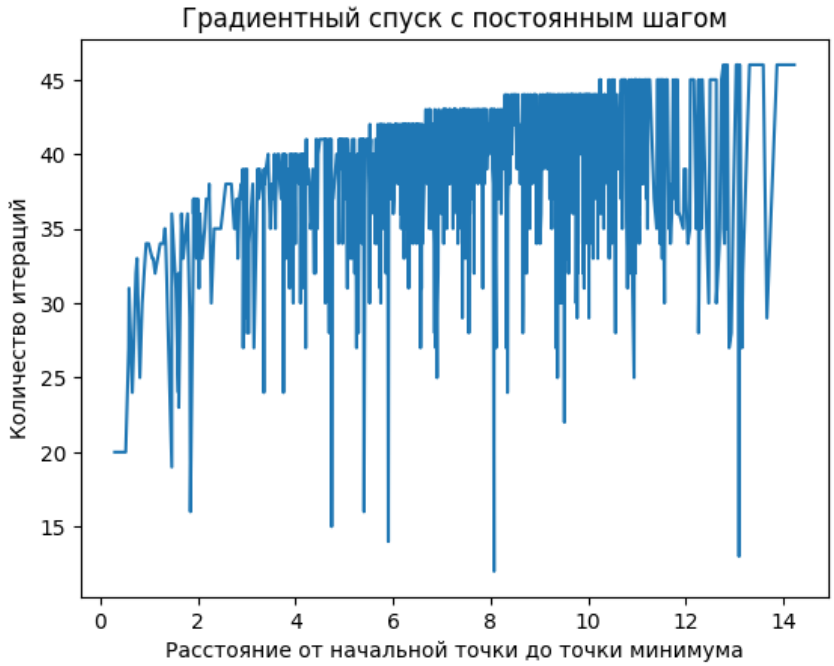
Выводы:

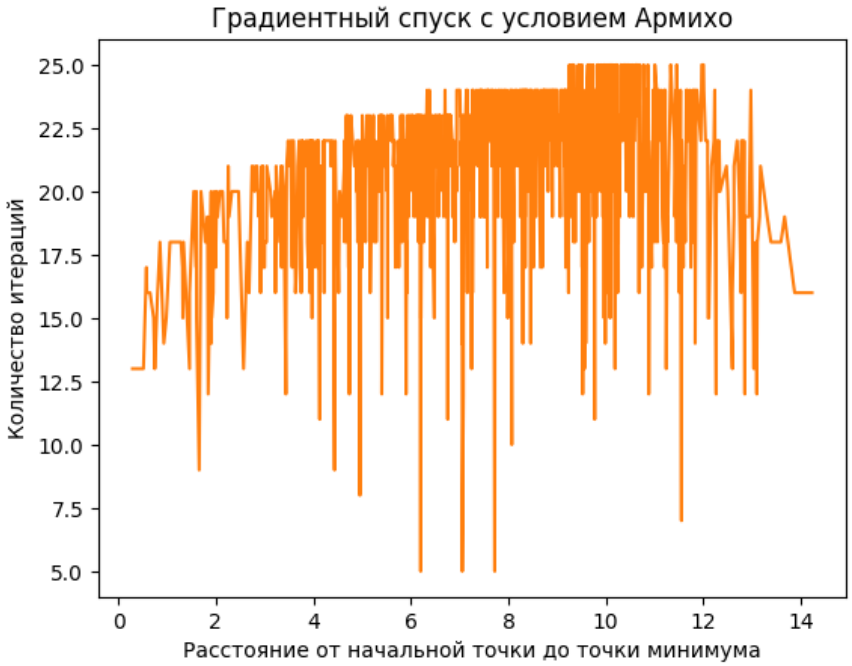
- Метод сопряженных градиентов всегда оказывается самым оптимальным на рассмотренных функциях.
- Метод наискорейшего спуска проигрывает методу сопряженных градиентов, но эффективнее других методов.
- Методы градиентного спуска с постоянным шагом и с дроблением шага по условию Армихо могут иметь разную относительную эффективность в зависимости от выбора функции.

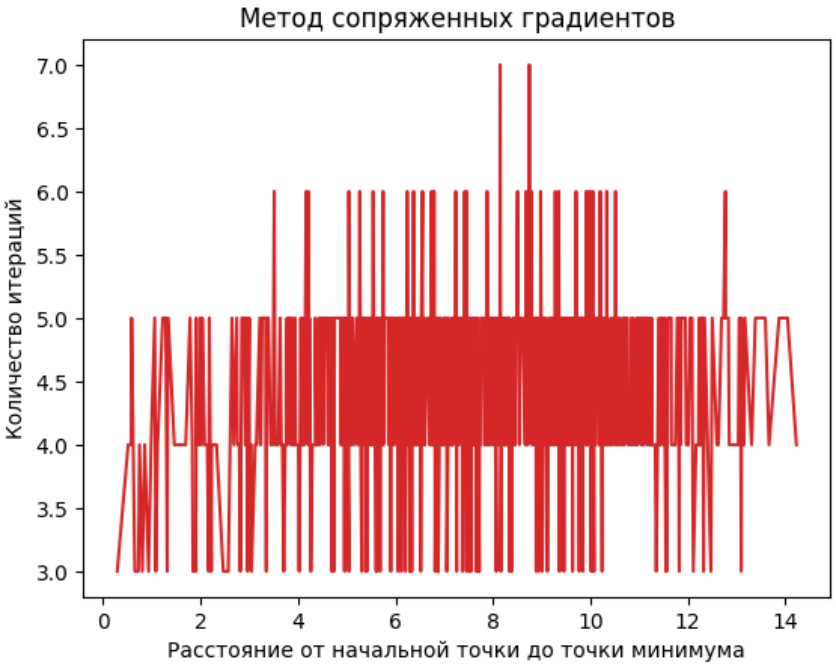
Зависимость числа итераций от выбора начальной точки

Рассмотрим то, как влияет выбор начальной точки на количество итераций каждого из методов:

1. **Функция** $f_1(x, y) = 6x^2 + 6xy + 6y^2 + 6x + 6y$**:**

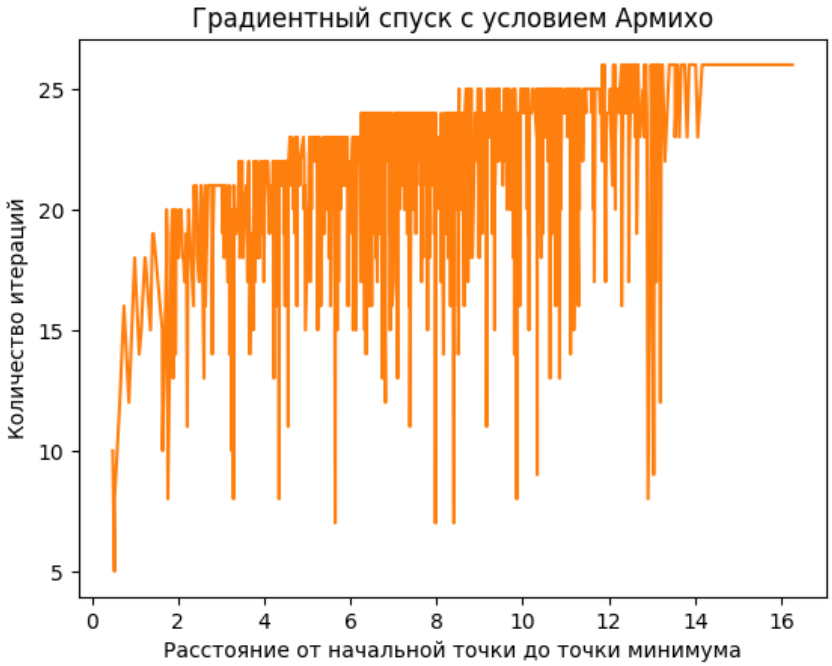

Градиентный спуск с постоянным шагом
Количество итераций
Расстояние от начальной точки до точки минимума
45
40
35
30
25
20
15
0
2
4
6
8
10
12
14

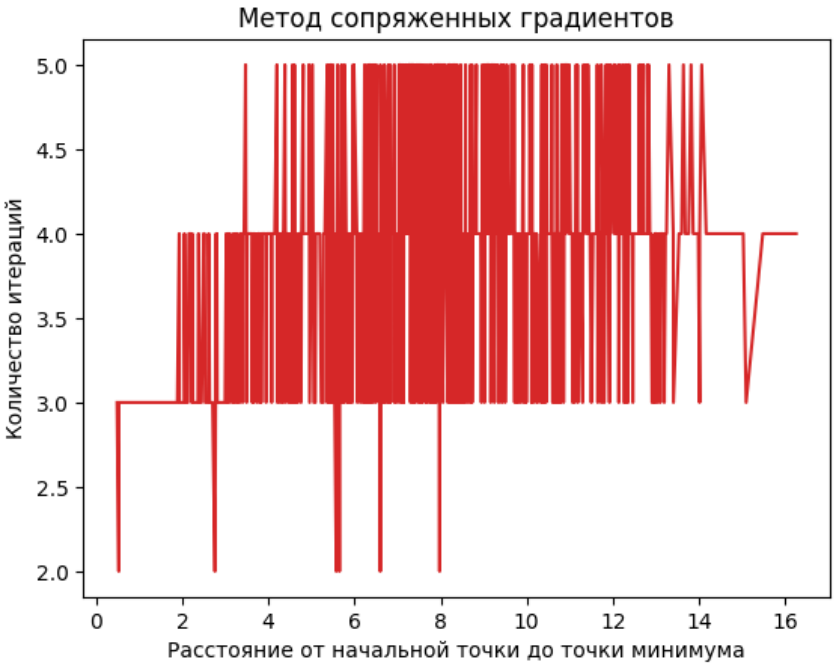

Градиентный спуск с условием Армихо
Количество итераций
Расстояние от начальной точки до точки минимума
25.0
22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
0
2
4
6
8
10
12
14



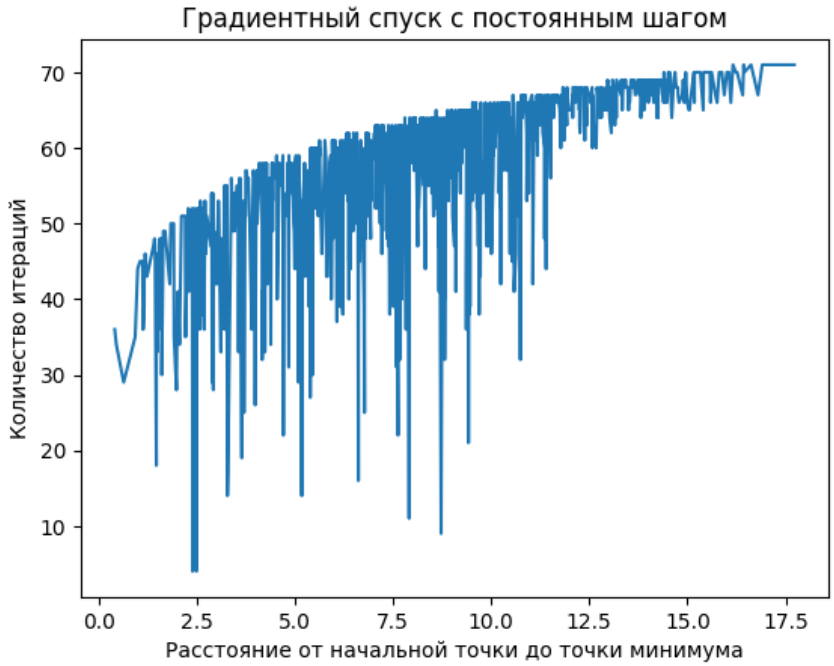
2. **Функция** $f_2(x, y) = 3x^2 - 2xy + 2y^2 + 5x + 5y + 10$:

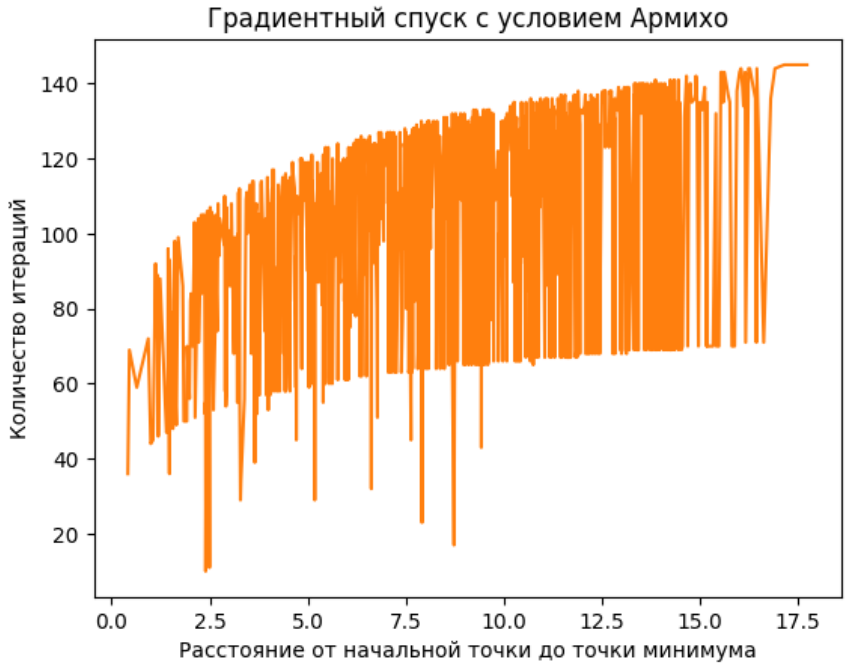

Градиентный спуск с постоянным шагом
Количество итераций
Расстояние от начальной точки до точки минимума
5
10
15
20
25
0
2
4
6
8
10
12
14
16

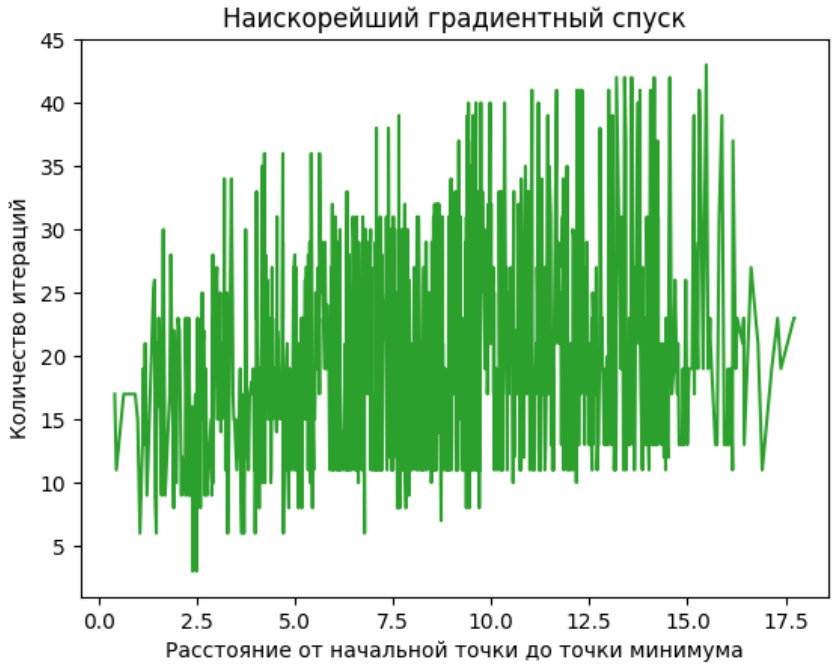

Градиентный спуск с условием Армихо
Количество итераций
Расстояние от начальной точки до точки минимума
5
10
15
20
25
0
2
4
6
8
10
12
14
16

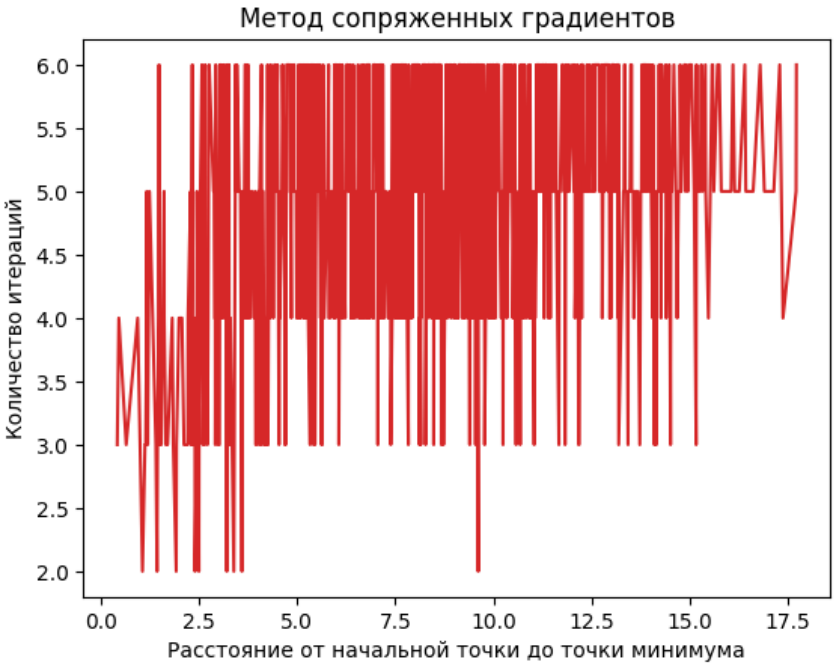


3. **Функция** $f_3(x, y) = 5x^2 + 3xy + y^2 + 7x + 7y$**:**


Градиентный спуск с постоянным шагом
Количество итераций
70
60
50
40
30
20
10
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Расстояние от начальной точки до точки минимума


Градиентный спуск с условием Армихо
Количество итераций
140
120
100
80
60
40
20
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Расстояние от начальной точки до точки минимума





Выводы:

- Методы градиентного спуска с постоянным шагом и с дроблением шага по условию Армихо имеют прямую корреляцию между расстоянием от начальной точки до минимума и числом итераций — чем больше расстояние, тем больше итераций потребуется.
- Метод наискорейшего спуска тоже имеет подобную корреляцию, однако менее заметную. Однако число итераций может варьироваться в несколько раз даже при одинаковом расстоянии между точками.

- Метод сопряженных градиентов работает весьма стабильно, и довольно тяжело говорить и зависимости между расстоянием от начальной точки до минимума и числом итераций — обычно этому методу нужно от 3 до 6 итераций для всех выбранных функций.

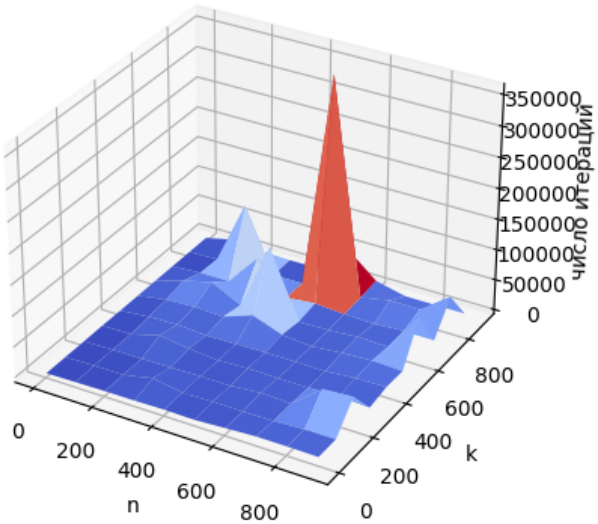
Генератор случайных квадратичных функций

Реализуем генератор случайных квадратичных функций от n перменных с заданным числом обусловленности k. Для упрощения задачи будем генерировать только диагональные матрицы A для функций вида $f(x) = \frac{1}{2}(Ax, x) + (b, x) + c$. Тогда число обусловленности функции будет равно отношению минимального и максимально числа на диагонали матрицы A (так как числа на диагонали будут являться собственными числами).

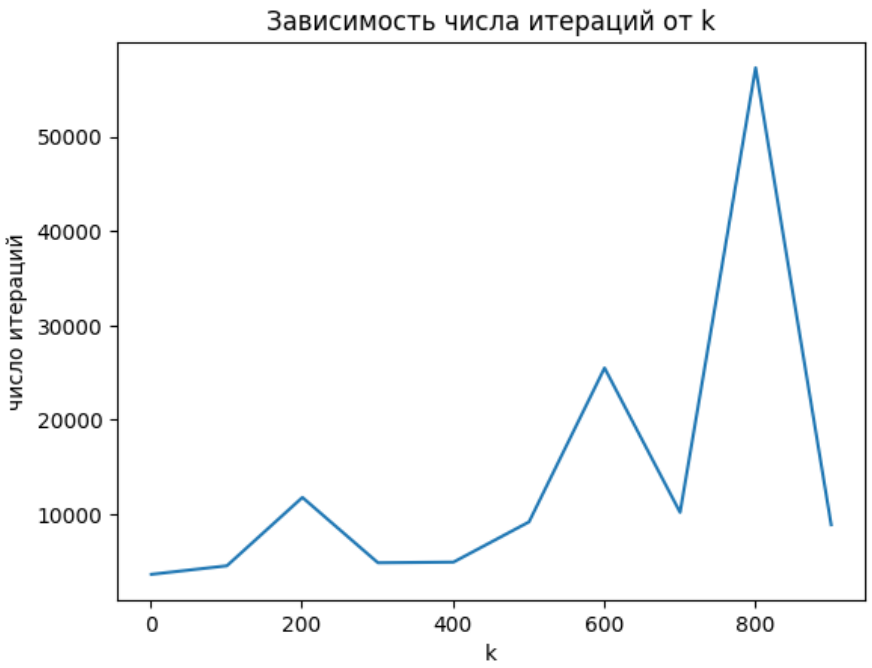
```
def generate_quadratic(dimensions, condition_number):
    min_ = np.random.rand()
    max_ = min_ * condition_number
    scale = max_ - min_
    a = np.random.rand(dimensions, dimensions) * scale
    a = np.diag(np.diag(a))
    min_pos = np.random.randint(0, dimensions)
    max_pos = np.random.randint(0, dimensions)
    while min_pos == max_pos:
        max_pos = np.random.randint(0, dimensions)
    a[min_pos, min_pos] = min_
    a[max_pos, max_pos] = max_
    b = np.random.rand(dimensions) * scale
    c = np.random.rand() * scale
    return QuadraticFunction(a, b, c)
```

Зависимость числа итераций градиентного спуска от размерности пространства и числа обусловленности

Так как для получения графика требуется немалое количество вычислений, все они были проведены всего лишь 1 раз:

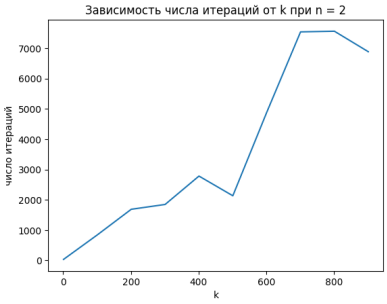


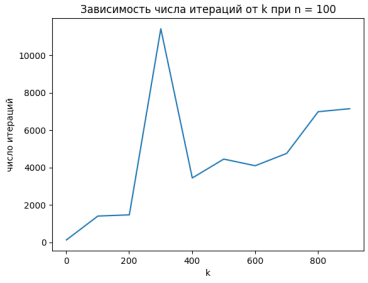
Если взять срез графика по осям k и числу итераций (и заменив значения при конкретном k средним значением) получим следующий график:

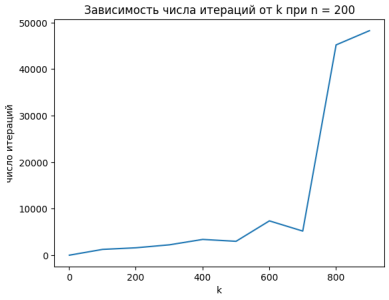


Даже несмотря на выбросы, можно сказать, что при возрастании k возрастает и число итераций.

Для примера вместо среднего значения можно привести несколько срезов относительно разных n:

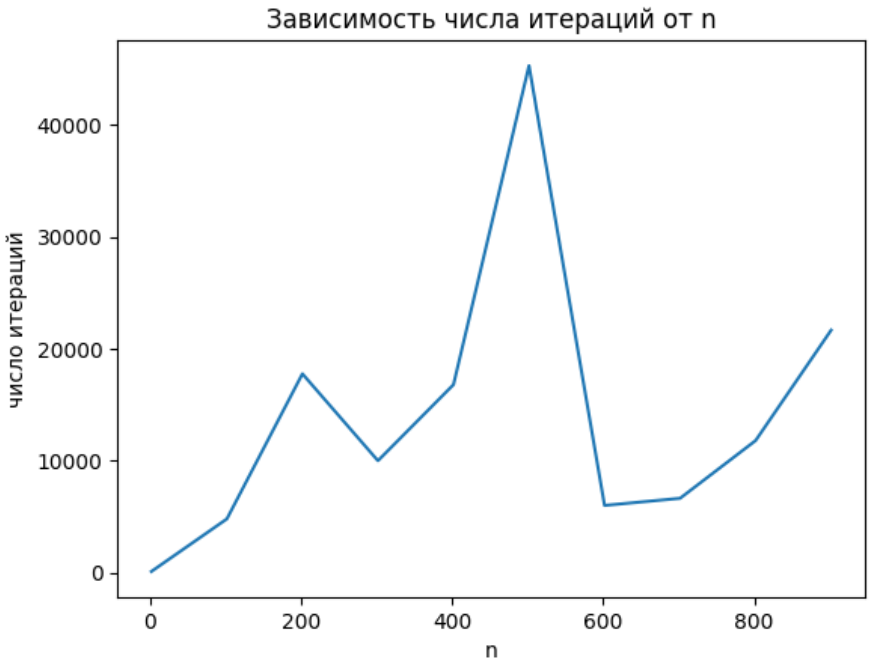






На всех этих графиках также заметна коррелаяция между k и количеством итераций.

Стоит отметить, что подобная зависимость не наблюдается для n (или является не такой явновыраженной):



Выводы:

- Число обусловленности функции сильно влияет на количество итераций — чем лучше обусловлена функция, тем меньше итераций потребуется градиентному спуску для завершения.
- Размерность пространства не влияет на число итераций метода.

Выводы

Были реализованы различные методы градиентного спуска и исследованы на квадратичных функциях. Было установлено, что самым эффективным по числу итераций является метод сопряженных градиентов. Самыми медленными оказались методы спуска с постоянным шагом и с дроблением шага по условию Армихо. Эти же методы имеют самую выраженную корреляцию между расстоянием от начальной точки до точки минимума и числом итераций (чем дальше начальная точка, тем больше итераций требуется этим методам).

Был реализован генератор квадратичных функций от n перменных с заданным числом обусловленности k. График зависимости числа итераций градиентного спуска от размерности пространства и числа обусловленности показал, что для хорошо обусловленных функций метод выполняется за меньшее число итераций, однако размерность пространства не влияет на количество итераций.