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

# Методы оптимизации

Отчёт по лабораторной работе № 1

**Работу выполнили:**
Ивченков Д. А., M32341
Султанов М. М., M32341
Трещёв А. С., M32341
**Преподаватель:**
Ким С. Е.

Санкт-Петербург
2023

**Цель работы:** изучение алгоритма градиентного спуска и методов его реализации, исследование его работы и характеристик.

**Задачи:**

1. Реализовать алгоритм градиентного спуска с постоянным шагом и методом одномерного поиска;
2. Исследовать работу алгоритма на различных функциях;
3. Проанализировать поведение и эффективность алгоритма;
4. Реализовать генератор случайных квадратичных функций;
5. Исследовать зависимость числа итераций алгоритма от размерности пространства и числа обусловленности функции;
6. Реализовать и исследовать алгоритм градиентного спуска с учётом условий Вольфе.

**Использованные библиотеки:**

1. NumPy;
2. Matplotlib;
3. numdifftools.

**Ссылка на реализацию:**

https://colab.research.google.com/drive/1naLxJz8WHFz_3nyOHumEFxp7WEWGMK1t?usp=sharing#scrollTo=egW6OFo5we7u

## Алгоритм градиентного спуска

Градиентный спуск – это метод линейного поиска, использующий градиенты функции. Он применяется при решении задачи поиска точки минимума гладкой функции $f(x)\colon \mathbb{R}^n \to \mathbb{R}$. Идея алгоритма заключается в оптимизации функции в направлении наискорейшего спуска, которое задаётся антиградиентом – вектором, противоположным градиенту функции $\nabla f(x)$. Спуск выполняется итерациями, на каждой из которых из текущей точки совершается шаг по направлению антиградиента.

$$x_{i+1} = x_i - \alpha \nabla f(x_i)$$

Величина шага определяется реализацией алгоритма: возможны методы постоянного шага, одномерного спуска, дробления шага и другие. Для завершения поиска точки минимума функции нужен критерий останова. Достаточным условием минимума является положительная определённость дифференциала второго порядка.

$$\nabla^2 f(x) > 0$$

Однако данный подход нерационален в силу сложности вычисления дифференциалов высших порядков функции. На практике используются более простые условия для нахождения приближённого значения минимума, такие как сравнение модуля разности двух последних точек или значений функции в них с какой-либо малой величиной.

$$\|x_{i+1} - x_i\| < \varepsilon$$

$$\|f(x_{i+1}) - f(x_i)\| < \varepsilon$$

Полученное в результате алгоритма значение – приближённое значение точки локального в общем случае минимума заданной функции.

## Метод постоянного шага

Данный метод подразумевает наличие шага постоянной величины. Спуск в таком случае на каждой итерации совершает шаги, пропорциональные длине градиента.

Реализация алгоритма приведена ниже. Функция const_step(f, x, epoch, paint_contour) принимает в качестве аргументов оптимизируемую функцию f, начальную точку x, максимальное количество итераций epoch, флаг paint_contour, необходимый для изображения траектории градиентного спуска, и величину постоянного шага learning_rate. В теле функции вычисляется градиент, далее в цикле происходит градиентный спуск с условием остановки на близость длины градиента к нулю $\|\nabla f(x_i)\| < \varepsilon$. Функция возвращает количество совершённых итераций и при наличии установленного флага выводит точку минимума и выводит изображение траектории градиентного спуска с линиями уровня функции.

```
gradient_eps = 1e-4


def const_step(f, x, epoch, paint_contour, learning_rate=0.09):
    grad = Gradient(f)
    n = len(x)
    points = np.zeros((epoch, n))
    points[0] = x
    count_of_iters = 0
    for i in range(1, epoch):
        gr = grad(x)
        if np.linalg.norm(gr) < gradient_eps:
            count_of_iters = i
            break
        x = x + learning_rate * (-gr)
        points[i] = x
    if paint_contour:
        print(x)
        points = points[:count_of_iters]
        plt.contour(X, Y, f([X, Y]), levels=sorted(list(set([f(p) for p in
points]))))
        plt.plot(points[:, 0], points[:, 1], '-r8')
    return count_of_iters
```

## Метод одномерного поиска

Метод одномерного поиска производит поиск минимума вдоль направления градиента функции с помощью какого-либо алгоритма поиска минимума на отрезке (метод дихотомии, метод Фибоначчи, метод золотого сечения) и делает шаг в найденную точку. Таким образом, шаг спуска оказывается различным на разных итерациях и зависит от текущей точки и поведения функции в её окрестности.

Реализация алгоритма приведена ниже. Алгоритм использует метод троичного поиска, реализованный в функции ternary_search(f, x, gr). Функция linear_search(f, x, epoch, paint_contour) принимает в качестве аргументов оптимизируемую функцию f, начальную точку x, максимальное количество итераций epoch, флаг paint_contour, необходимый для изображения траектории градиентного спуска. В теле функции вычисляется градиент, после этого происходит градиентный спуск с условием останова, идентичным условию в первом методе. Функция возвращает количество итераций, затраченное на поиск минимума, и при наличии установленного флага печатает точку минимума заданной функции и выводит изображение траектории градиентного спуска с линиями уровня функции.

```
eps = 1e-4


def ternary_search(f, x, gr):
    left = 0
    right = 10
    while right - left > eps:
        a = (left * 2 + right) / 3
        b = (right * 2 + left) / 3
        if f(x - a * gr) < f(x - b * gr):
```

```
            right = b
        else:
            left = a
    return (left + right) / 2


def linear_search(f, x, epoch, paint_contour):
    grad = Gradient(f)
    points = np.zeros((epoch, 2))
    points[0] = x
    count_of_iters = 0
    for i in range(1, epoch):
        gr = grad(x)
        if np.linalg.norm(gr) < gradient_eps:
            count_of_iters = i
            break
        t = ternary_search(f, x, gr)
        x = x + t * (-gr)
        points[i] = x
    if paint_contour:
        print(x)
        points = points[:count_of_iters]
        plt.contour(X, Y, f([X, Y]), levels=sorted(list(set([f(p) for p in
points])))))
        plt.plot(points[:, 0], points[:, 1], '-r8')
    return count_of_iters
```

## Исследование методов на примере квадратичных функций

Для исследования работы алгоритмов и их свойств были взяты две квадратичные функции $f$ и $g$ от двух аргументов, на которых поведение методов различается.

$$f(x, y) = x^2 - 2xy + y^2 + 4x - 4y + 5$$

$$g(x, y) = 11(x - 1.25)^2 + (y - 6.9)^2$$

Функция $f$ задаёт параболический цилиндр и имеет множество точек минимума, расположенное на прямой $y - x = 2$.

Функция $g$ задаёт эллиптический параболоид с точкой минимума $(1.25, 6.9)$.

Графики функций $f$ и $g$ изображены на рисунках 1 и 2 соответственно.

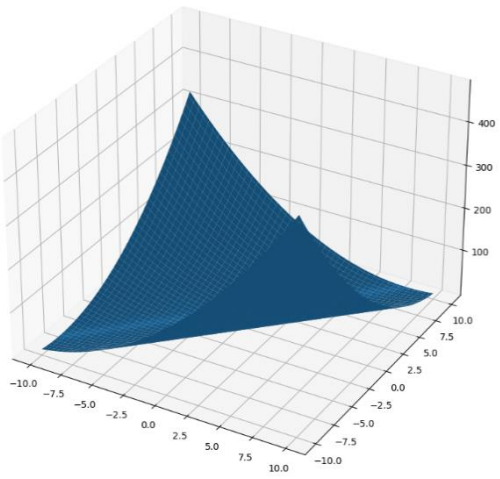


*Рисунок 1 – график f(x, y)*

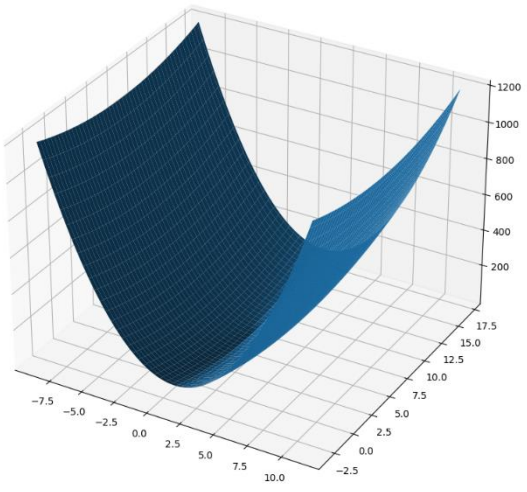


*Рисунок 2 – график g(x, y)*

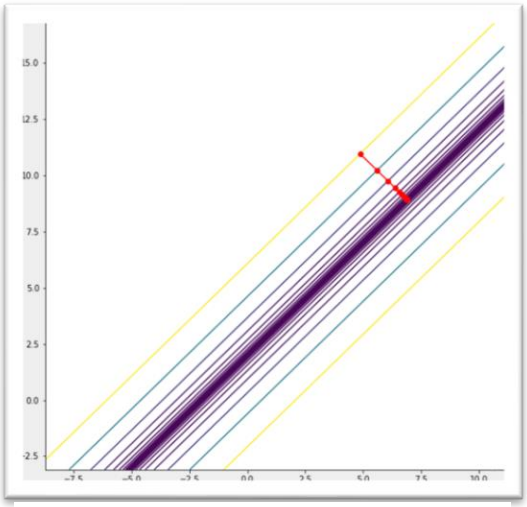

*Рисунок 3 – Траектория градиентного спуска с постоянным шагом для f(x, y)*

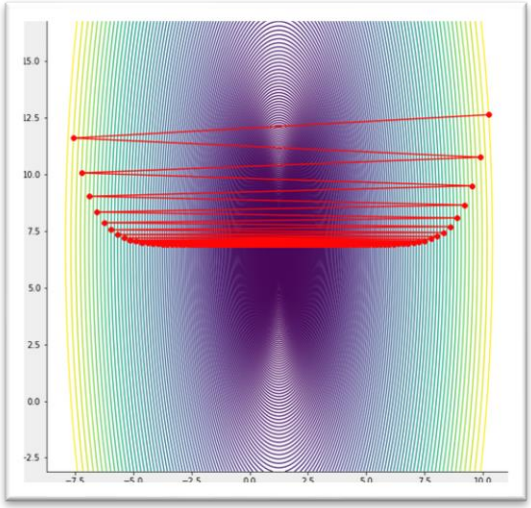

*Рисунок 4 – Траектория градиентного спуска с постоянным шагом для g(x, y)*

## (a) Метод постоянного шага

На рисунках 3 и 4 показаны возможные траектории градиентного спуска с постоянным шагом для функций $f$ и $g$ соответственно. Для первой функции алгоритм отработал гораздо быстрее, чем для второй. Хорошо заметно, что данный метод сильно зависит от выбранной функции и начальной точки, а также от величины шага, и, вообще говоря, может сходиться крайне медленно. Если функция убывает плавно, то её градиент имеет малую величину, следовательно спуск будет происходить короткими шагами и для достижения точки минимума понадобится внушительное число итераций. Если же градиент принимает большие значения, т.е. склон функции достаточно резкий, возникнет другая проблема: шаги алгоритма могут быть длинными настолько, что точки будут выбираться на разных сторонах склона, тем самым перемещаясь около минимума, но сходясь к нему долгим путём. Описанное явление чётко прослеживается на рисунке 4.

## (b) Метод одномерного поиска

Примеры работы градиентного спуска с одномерным поиском представлены на рисунках 5 и 6 для функций $f$ и $g$ соответственно.

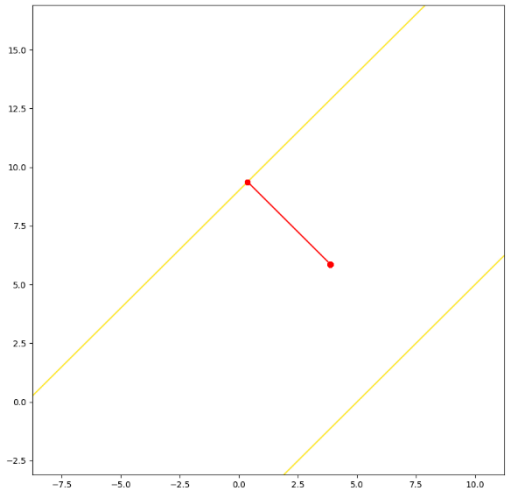


*Рисунок 5 – Траектория градиентного спуска с одномерным поиском для функции f(x, y)*

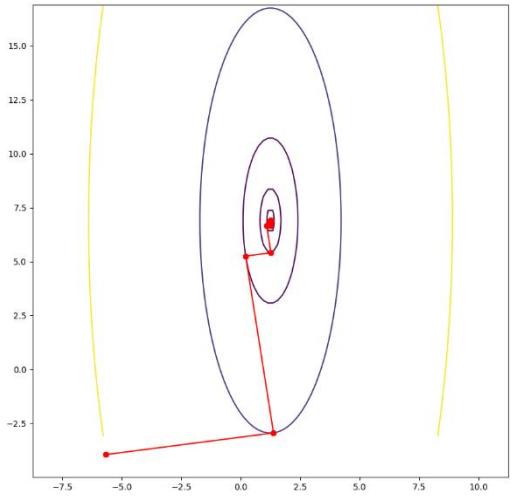


*Рисунок 6 – Траектория градиентного спуска с одномерным поиском для функции g(x, y)*

Алгоритм выполнил задачу для функции $f$ быстрее, чем для $g$. Причём в первом случае для нахождения минимума ему хватило буквально одной итерации. Во второй ситуации спуск произвёл некоторое большее количество запусков, но все ещё сошёлся значительно быстрее метода постоянного шага. Такое отличие объясняется тем, что данный алгоритм выбирает шаг более рационально. Находясь в очередной точке, он ищет минимум на направлении антиградиента, таким образом за одну итерацию перемещаясь в наиболее подходящую для продолжения спуска точку и пропуская промежуточные шаги, которые совершает метод с постоянным шагом. Однако у одномерного поиска есть схожие с ранее рассмотренными недостатки. В случае если функция пологая, в силу малого значения градиента метод будет способен рассмотреть только отрезок небольшой длины, найдя возможно не самое оптимальное для спуска место. Зато при быстрых изменениях функции эффективность поиска следующей точки не уменьшается.

На основании множественных замеров и их усреднения произведено сравнение работы двух методов градиентного спуска с точки зрения количества вычислений минимизируемой функции и ее градиентов. Полученные средние значения представлены в таблице 1.

*Таблица 1 – Среднее количество вычислений функции и её градиента для методов градиентного спуска и функций f и g*

| | $f(x,y)$ | $g(x,y)$ |
|---|---|---|
| Метод постоянного шага | 29.27 | 706.68 |
| Метод одномерного поиска | 2.85 | 15.51 |

Опираясь на данные таблицы, можно сделать вывод, что метод одномерного поиска во многих случаях сходится заметно быстрее и вычисляет значения функций реже, чем метод постоянного шага.

## (c) Зависимость работы методов в зависимости от выбора начальной точки

В этом пункте было исследовано, как ведёт себя градиентный спуск для функций $f$ и $g$ в зависимости от выбора начальной точки запуска градиентного спуска.

1. Для функции $f$:

Начальная точка x = (0; 0) (рис. 7)

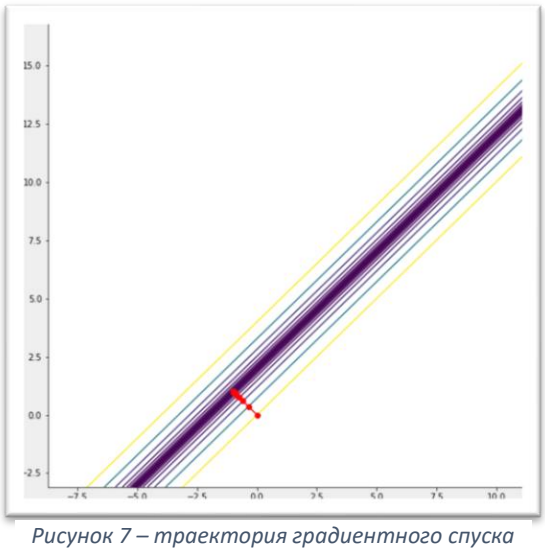

*Рисунок 7 – траектория градиентного спуска для начальной точки (0; 0)*

Начальная точка x = (10; 0) (рис. 8)

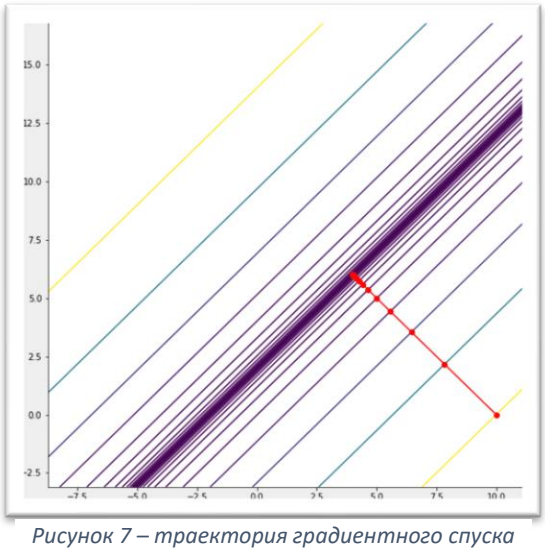

*Рисунок 7 – траектория градиентного спуска для начальной точки (10; 0)*

Начальная точка x = (-7.5; 15)  (рис. 9)

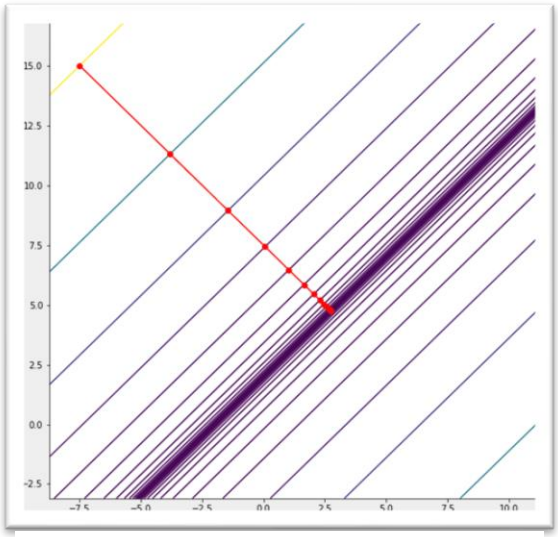

*Рисунок 9 – траектория градиентного спуска для начальной точки (-7.5; 15)*

Начальная точка x = (-1.5; 0) (рис. 10)

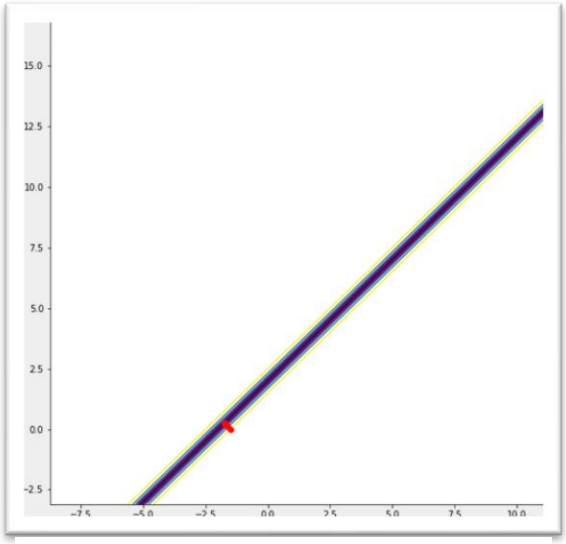

*Рисунок 10 – траектория градиентного спуска для начальной точки (-1.5; 0)*

Как видно из траекторий, чем дальше от точки минимума запускаем градиентный спуск, тем больше итераций и больше времени требуется, чтобы минимум был найден

2. Для функции $g$:

Начальная точка x = (0; 0) (рис. 11)

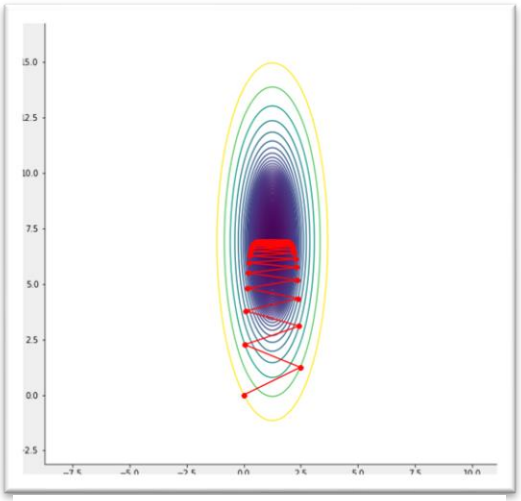

*Рисунок 11 - траектория градиентного спуска для начальной точки (0; 0)*

Начальная точка x = (1.25; 15) (рис. 12)

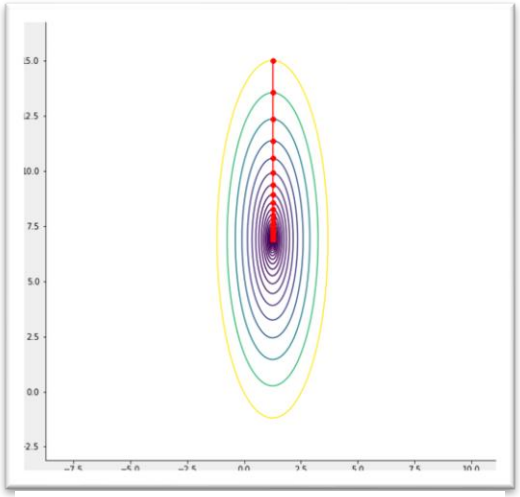

*Рисунок 12 – траектория градиентного спуска для начальной точки (1.25; 15)*

Начальная точка x = (10; 6.9) (рис. 13)

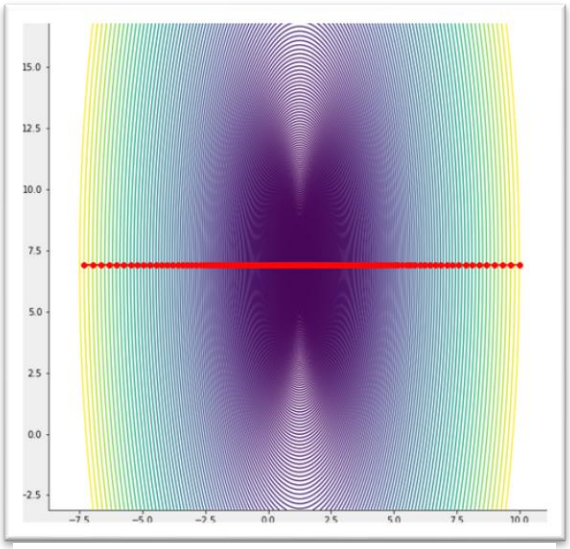

*Рисунок 13 – траектория градиентного спуска для начальной точки (10; 6.9)*

Начальная точка x = (10; 15) (рис. 14)

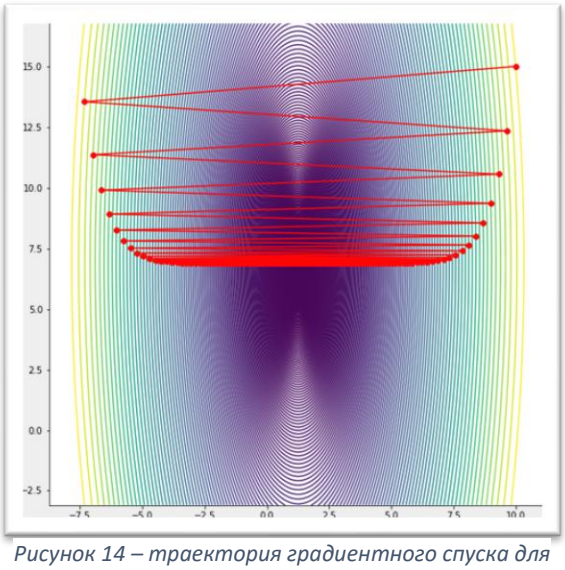

*Рисунок 14 – траектория градиентного спуска для начальной точки (10; 15)*

Как видно из графиков, для функции $g$ также справедливо утверждение, что чем дальше от точки минимума запускаем градиентный спуск, тем больше итераций требуется для нахождения минимума, однако функция такова, что при выборе начальной точки ближе к 1.25 по оси Ox (минимум функции) шаг градиентного спуска меньше промахивается и меньше переходит на противоположный склон, что случается при выборе начальной точки ближе к 6.9 по оси Oy (минимум функции). Это связано с тем, что функция очень вытянута вдоль одной из осей, поэтому и получается значительно больше шаг градиентного спуска.

## (d) Влияние нормализации на сходимость градиентного спуска

В данном пункте проверяется, как зависит работа градиентного спуска в зависимости от сжатия/растяжения графика одной и той же функции по осям координат Ox и Oy. Была взята на рассмотрение функция $g(x, y) = \alpha(x - 1.25)^2 + \beta(y - 6.9)^2$

Заметим, что у такой функции сжатие/растяжение графика по оси Ox определяется коэффициентом $\alpha$, а по оси Oy – коэффициентом $\beta$.

1. $\alpha = 0.1,\ \beta = 1$ (рис. 15 и 16)

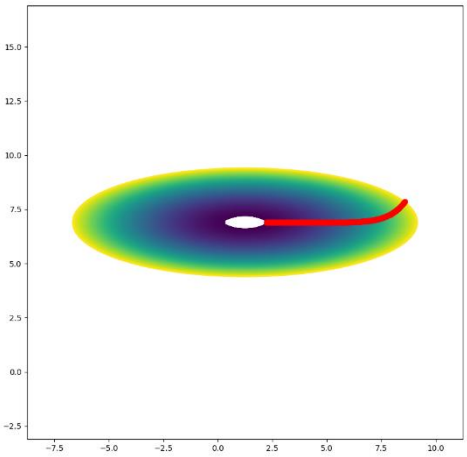

*Рисунок 15 – траектория градиентного спуска с постоянным шагом для коэффициентов $\alpha = 0.1,\ \beta = 1$*

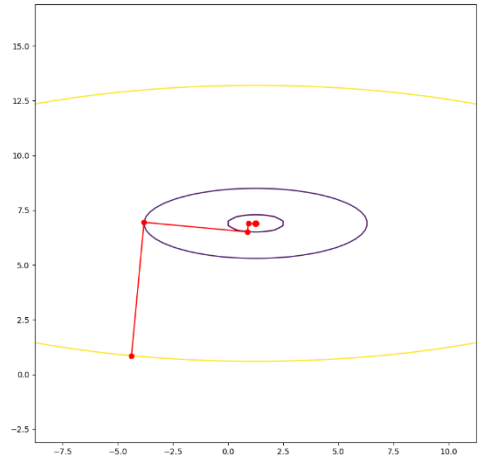

*Рисунок 16 – траектория градиентного спуска с одномерным поиском для коэффициентов $\alpha = 0.1,\ \beta = 1$*

2. $\alpha = 1,\ \beta = 10$ (рис. 17 и 18)

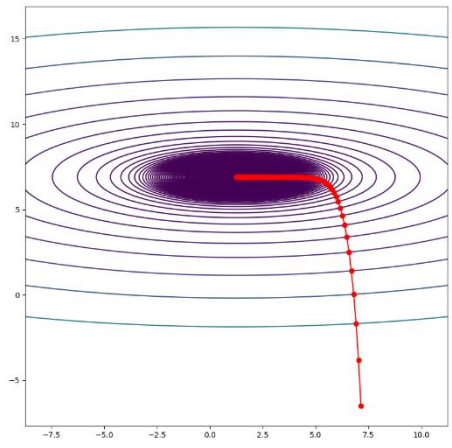

*Рисунок 17 – траектория градиентного спуска с постоянным шагом для коэффициентов $\alpha = 1,\ \beta = 10$*

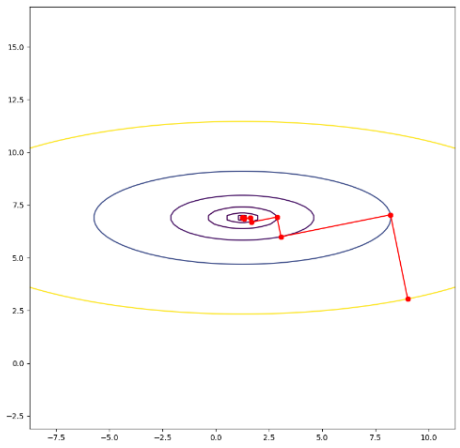

*Рисунок 18 – траектория градиентного спуска с одномерным поиском для коэффициентов $\alpha = 1,\ \beta = 10$*

3. $\alpha = 0.01,\ \beta = 10$ (рис. 19 и 20)

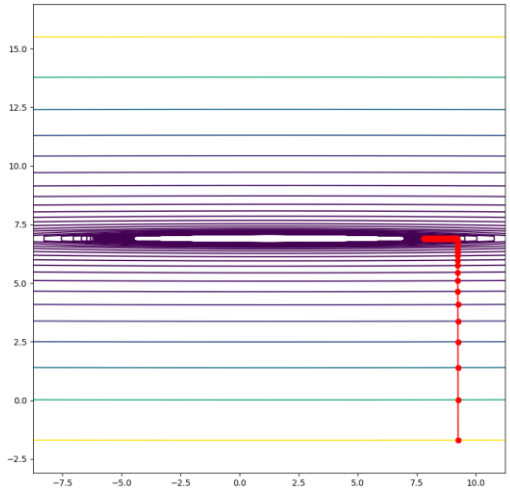


*Рисунок 19 – траектория градиентного спуска с постоянным шагом для коэффициентов* $\alpha = 0.01,\ \beta = 10$

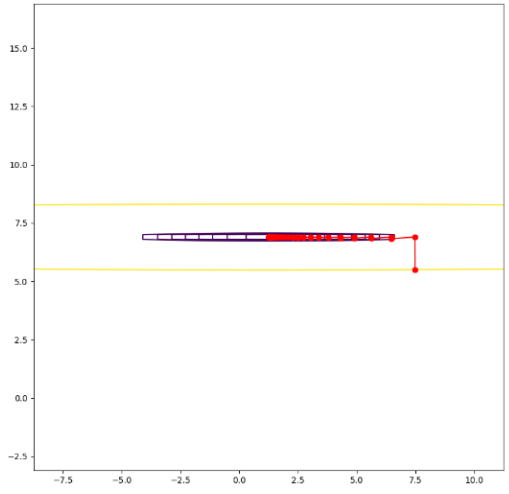


*Рисунок 20 – траектория градиентного спуска с одномерным поиском для коэффициентов* $\alpha = 0.01,\ \beta = 10$

4. $\alpha = 10,\ \beta = 10$ (рис. 21 и 22)

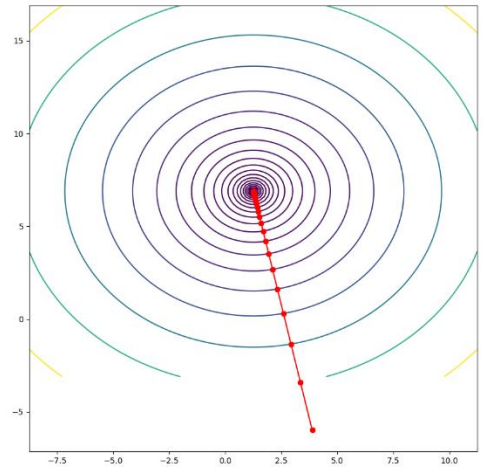


*Рисунок 21 – траектория градиентного спуска с постоянным шагом для коэффициентов* $\alpha = 10,\ \beta = 10$

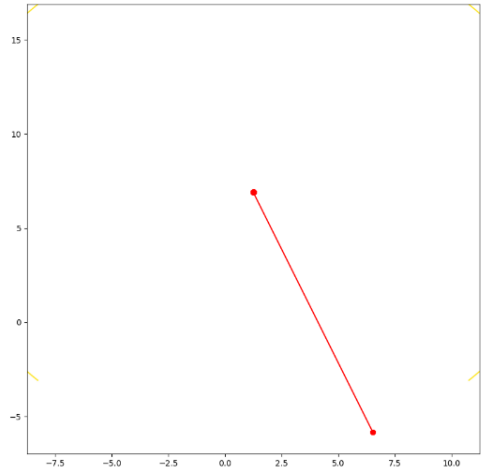


*Рисунок 22 – траектория градиентного спуска с одномерным поиском для коэффициентов* $\alpha = 10,\ \beta = 10$

5. $\alpha = 0.01,\ \beta = 0.1$ (рис. 23 и 24)

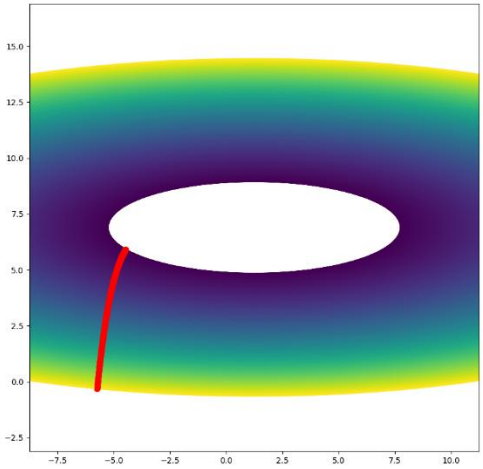


*Рисунок 23 – траектория градиентного спуска с постоянным шагом для коэффициентов* $\alpha = 0.01,\ \beta = 0.1$

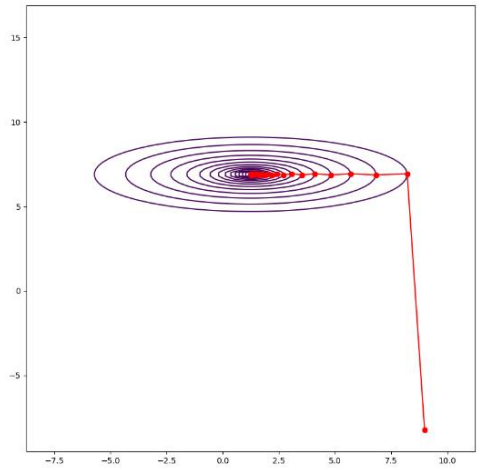


*Рисунок 24 – траектория градиентного спуска с одномерным поиском для коэффициентов* $\alpha = 0.01,\ \beta = 0.1$

6. $\alpha = 0.001,\ \beta = 0.01$ (рис. 25 и 26)

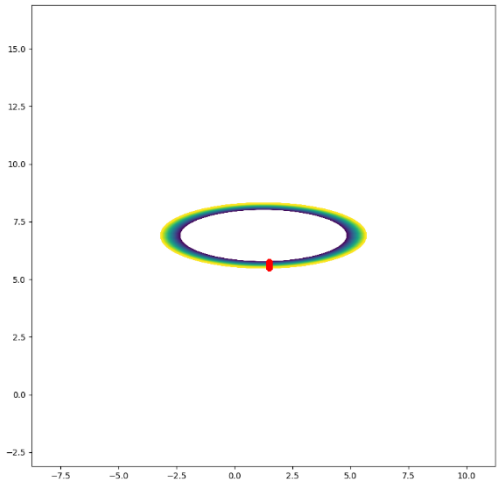


*Рисунок 25 – траектория градиентного спуска с постоянным шагом для коэффициентов* $\alpha = 0.001,\ \beta = 0.01$

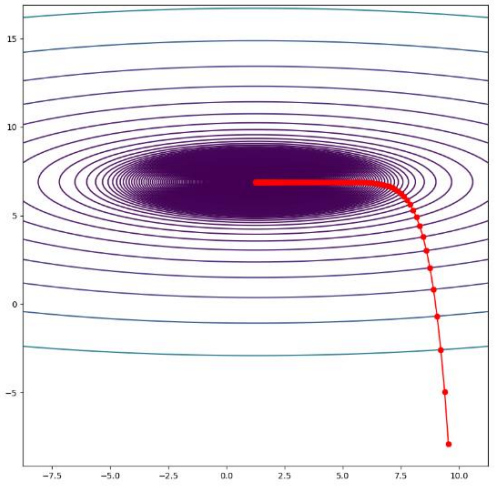


*Рисунок 26 – траектория градиентного спуска с одномерным поиском для коэффициентов* $\alpha = 0.001,\ \beta = 0.01$

## Генератор случайных квадратичных функций

Любая квадратичная функция $f: \mathbb{R}^n \to \mathbb{R}$ имеет вид

$$f(x) = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}x_i x_j + \sum_{i=1}^{n} b_i x_i + c$$

где $x \in \mathbb{R}^n$, $a_{ij}, b_i, c \in \mathbb{R}$,

т.е. такая функция представима в виде суммы квадратичной формы, вектора и константы, следовательно, для генерации произвольной квадратичной функции достаточно генерировать эти три сущности.

$$f(x) = x^T A\, x + \, b^T x + c$$

Матрица квадратичной формы должна иметь заданное число обусловленности – степень чувствительности функции к изменениям аргумента. Число обусловленности матрицы тесно связано с её сингулярными числами.

Сингулярное разложение невырожденной квадратной матрицы $A$ – это её декомпозиция вида

$$A = U\,\Lambda\,V^T$$

где $U$ и $V$ – вещественные ортогональные матрицы, $\Lambda = diag(\lambda_1, \dots, \lambda_n)$ – вещественная диагональная матрица, $\lambda_1 \geq \cdots \geq \lambda_n$ – сингулярные числа $A$.

Определение числа обусловленности матрицы $\mu(A) = \|A\| \cdot \|A^{-1}\|$ зависит от выбора нормы. Пусть выбрана евклидова норма матрицы

$$\|A\| = \max_{|x|=1} \left(\frac{|Ax|}{|x|}\right)$$

Тогда евклидову норму матрицы несложно вычислить, если известны её сингулярные числа.

$$\|A\| = \lambda_1$$

$$\|A^{-1}\| = \frac{1}{\lambda_n}$$

Из перечисленных фактов следует следующая формула числа обусловленности матрицы $A$

$$\mu(A) = \frac{\lambda_1}{\lambda_n}$$

Таким образом, чтобы получить случайную матрицу с заданным числом обусловленности $k \geq 1$, необходимо задать её сингулярные числа $\lambda_1 = k \geq \cdots \geq \lambda_1 = 1$ и умножить $D = diag(\lambda_1, \dots, \lambda_n)$ слева и справа на случайные ортогональные $U$ и $V$.

Для генерации случайной квадратичной функции нужна случайная симметричная матрица, описывающая квадратичную форму. $D = diag(\lambda_1, \dots, \lambda_n)$ является симметричной и в силу того, что $\lambda_i \geq 0$, задаёт положительно определённую форму, т.е. функцию, имеющую минимум. Согласно закону инерции квадратичных форм, сигнатура формы (и, следовательно, её знакоопределённость) не изменяется при невырожденных преобразованиях, приводящих её к каноническому (диагональному) виду. Тогда для любой невырожденной матрицы $Q$ форма, задающаяся матрицей $A = Q^{-1}DQ$, так же является положительно определённой. В случае ортогональной матрицы $Q^{-1} = Q^T$ получаем

$$A = Q^T DQ$$

Более того, $A$ – симметричная:

$$A^T = (Q^T DQ)^T = Q^T DQ = A$$

Значит, матрица $A$ задаёт удовлетворяющую требованиям квадратичную форму. Таким образом, предоставлен алгоритм генерации случайной квадратичной функции с числом обусловленности $k$. Матрица $D$ генерируется как диагональная с первым и последним элементами, равными k и 1 соответственно. Остальные числа на диагонали между первым и последним не возрастают. Ортогональная матрица $Q$ получается как результат QR-разложения случайной матрицы. Дополнительно генерируются случайный вектор и случайная константа. Функция генерации generate_quadratic_function(n, k) принимает в качестве аргументов размерность n и число обусловленности k, возвращает лямбда-выражение, соответствующее случайной квадратичной функции. Реализация алгоритма представлена ниже.

```
def generate_quadratic_function(n, k):
    Q, R = np.linalg.qr(np.random.rand(n, n))
    A = np.diag(np.linspace(k, 1, n))
    A = np.dot(Q.T, np.dot(A, Q))
    b = np.random.rand(n)
    c = np.random.rand()
    return lambda x: np.dot(x, np.dot(A, x)) + np.dot(b, x) + c
```

С помощью реализованного генератора случайных квадратичных функций была исследована зависимость $T(n, k)$ числа итераций, необходимых градиентному спуску для

сходимости в зависимости от размерности пространства $n$ и числа обусловленности оптимизируемой функции $k$. Полученный график представлен на рисунке 27.

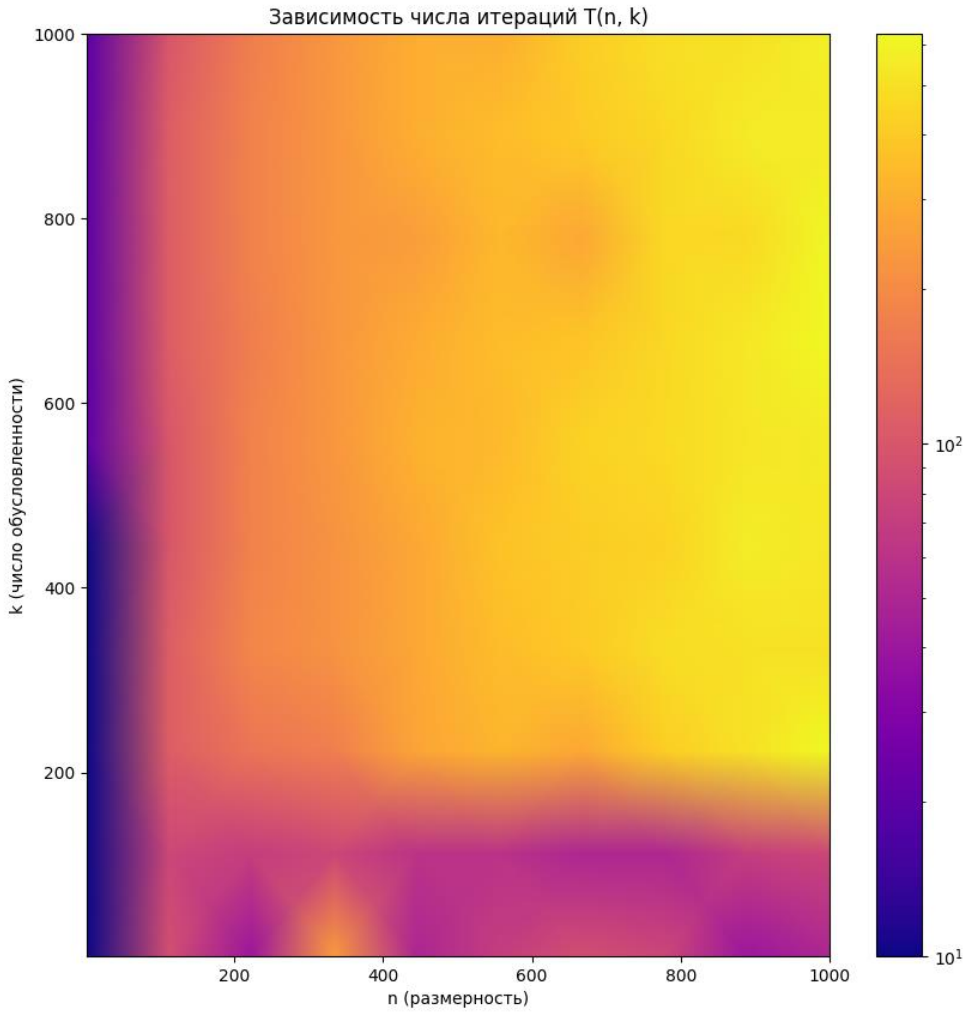


*Рисунок 27 – график зависимости числа итераций T(n, k) от размерности пространства n и числа обусловленности k*

Анализ распределения цвета на графике позволяет понять, что при хорошей обусловленности функции, т.е. при значениях $k$, близких к единице, градиентный спуск сходится быстрее, чем в случаях плохо обусловленных функций (при больших $k$). Кроме того, наблюдается некоторая зависимость от размерности пространства: при небольших размерностях градиентный спуск работает значительно быстрее других случаев, что вероятно связано с более сложным поведением функций при увеличении размерности пространства.

## Алгоритм градиентного спуска с учётом условий Вольфе

Условия Вольфе – это набор условий, которые используются для того, чтобы реализовывать приближённый поиск вдоль направления. Нас конкретно интересует условие Армаджио, также известное как условие достаточное убывание:

$$f(x_k + t_k p_k) \leq f(x_k) + \mathrm{c_1 t_k} \nabla\big(\mathrm{f(x_k)}\big)^{\mathrm{T}} p_k$$

При выполнении этого условия остаются только те аргументы, которые имеют значение меньше нежели функция, указанная в правой части, которая в свою очередь является просто прямой.

Помимо условия Армаджио нам нужно выполнение условия кривизны, для того чтобы точка минимума существовала. Снизу приведены условие кривизны и сильное условие для кривизны:

$$\nabla f(x_k + t_k p_k)^T p_k \geq c_2 \nabla f(x_k)^T p_k$$

$$|\nabla f(x_k + t_k p_k)^T p_k| \leq c_2 |\nabla f(x_k)^T p_k|$$

Данное условие задаёт, по сути, ограничение на то, чтобы скорость изменения функции в следующей точке было не больше текущей. Перейдём теперь к реализации.

Воспользуемся методом одномерного поиска, который был написан до этого, но теперь на каждом шаге тернарного поиска мы будем проверять, удовлетворяет ли точка условиям Вольфе. И если точка удовлетворяет этим условиям, то мы считаем её следующей, а иначе ищем дальше точку так же, как и до этого тернарным поиском. Реализация алгоритма ниже:

<Вставьте здесь код пж>

В чём же всё-таки преимущество выполнение условий Вольфе? В том, что нам теперь не надо вычислять минимум на направлении максимально точно, а можем выйти из тернарника гораздо раньше, а диапазон дозволенных аргументов мы задаём коэффициентами

$c_1$ и $c_2$, которые варьируются от 0 до 1. И чем ближе они к нулю, тем более точный результат мы получаем. Посмотрим на реализацию этого алгоритмы и результаты в зависимости от разного параметра $c_1$ и $c_2$. Для начала сгенерируем функцию с числом обусловленности 100, а затем запустим обычный линейный поиск, линейный поиск с параметрами c1=c2=1e-8 и 1e-3.

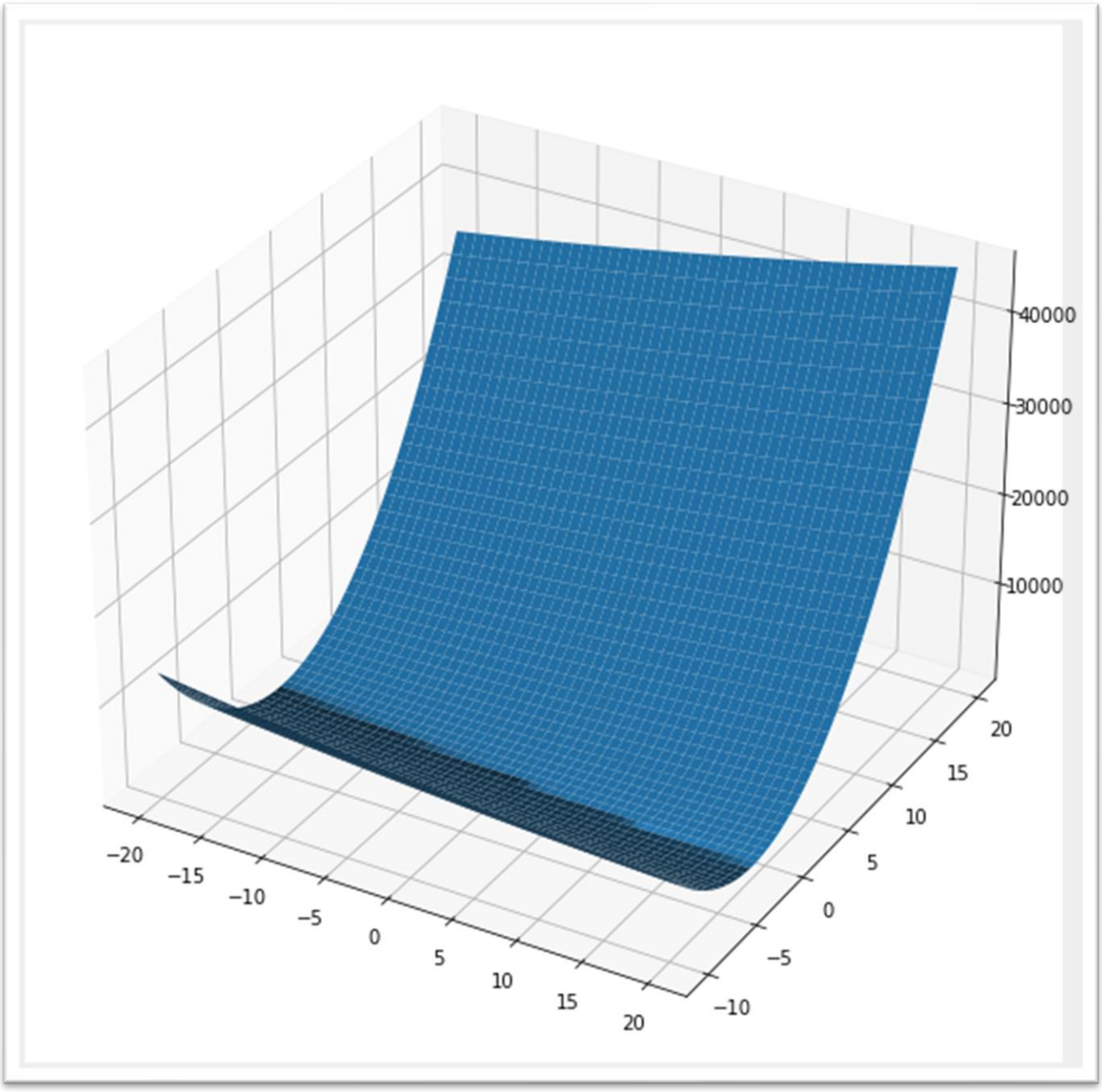


*Рисунок 28*

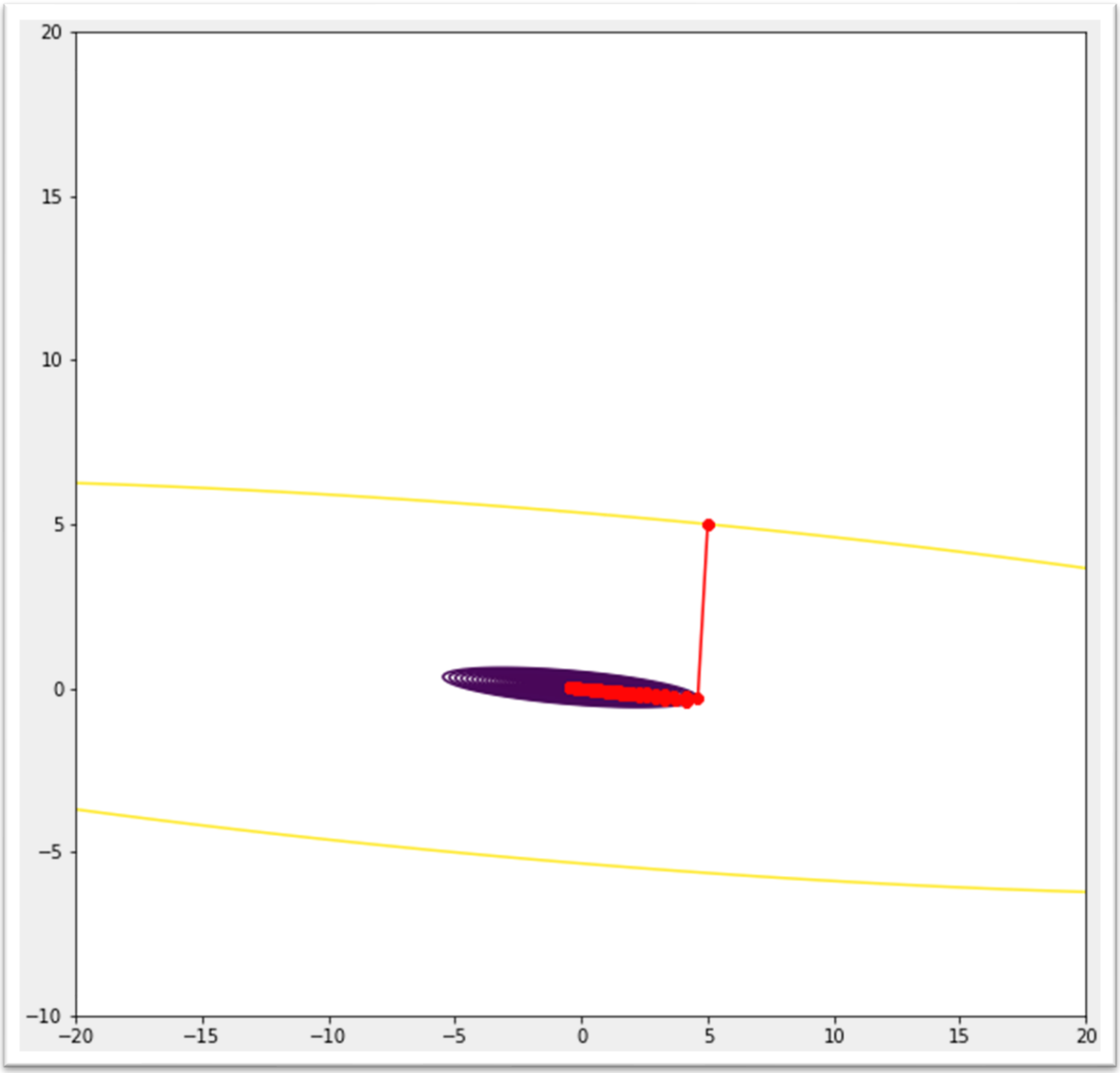


*Рисунок 29*

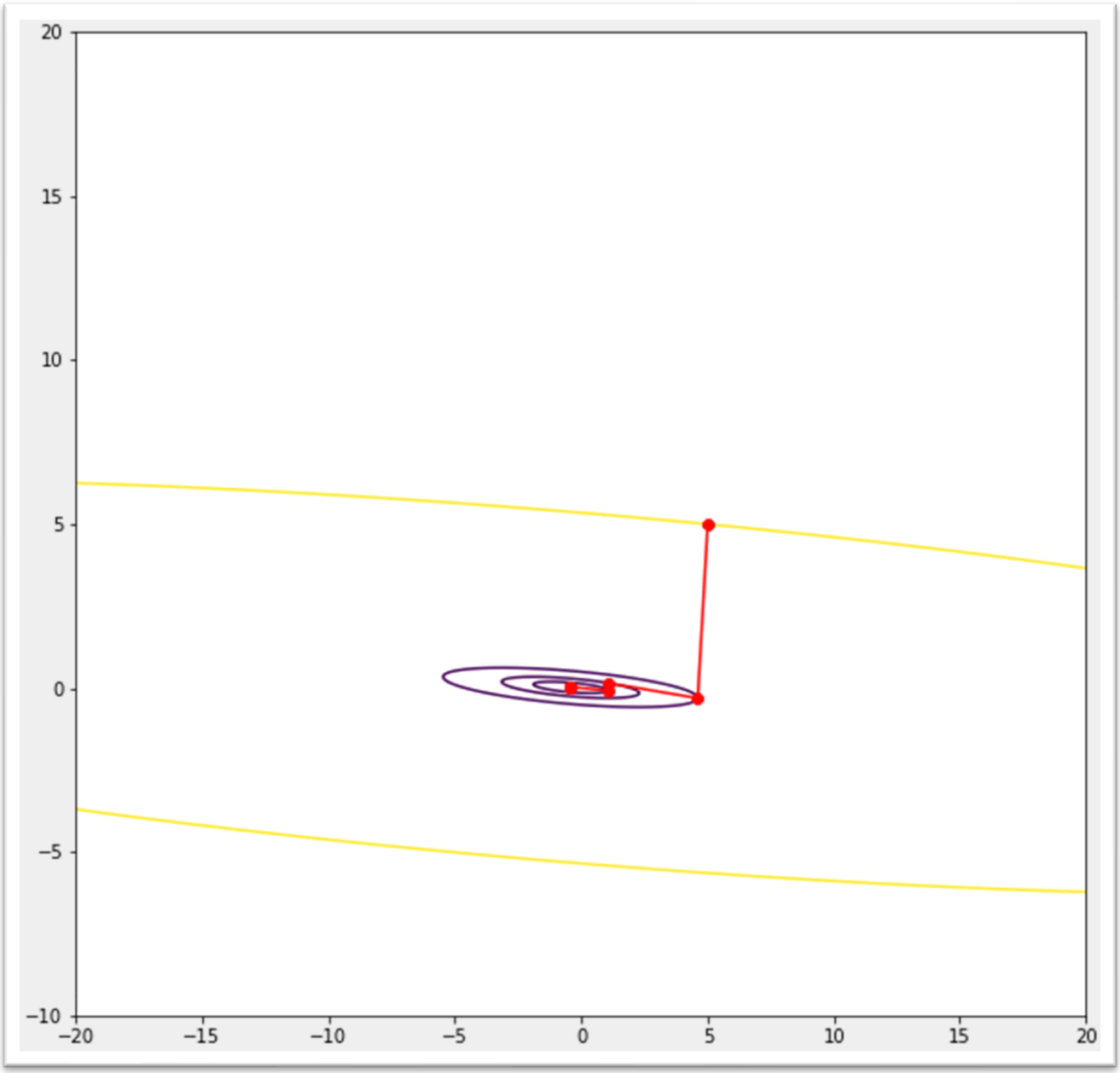


*Рисунок 30*

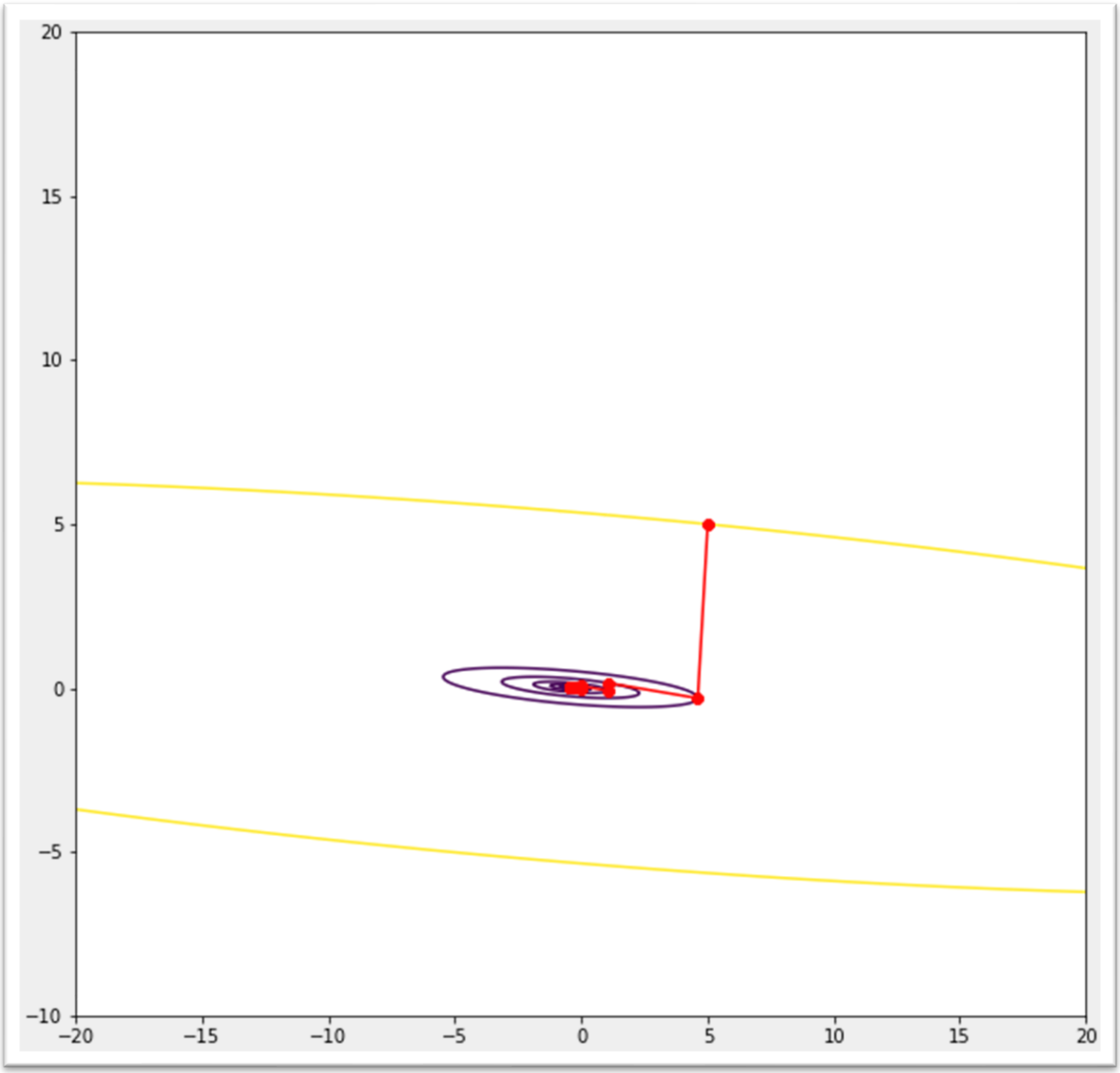


*Рисунок 31*

Как можно заметить, функция градиентного спуска с использованием условия Вольфе может работать быстрее, чем одномерный поиск

# Выводы

Алгоритм градиентного спуска позволяет решать задачу нахождения точки минимума гладкой функции. Возможны различные реализации алгоритма. В работе были рассмотрены метод постоянного шага и метод одномерного поиска. На квадратичных функциях метод одномерного поиска проявляет заметно лучшую сходимость в сравнении с методом постоянного шага. У обоих вариантов в ходе исследования обнаружились недостатки: сильная зависимость от поведения функции и начальной точки, эмпирический подбора неизменяемых параметров, наложенные ограничения на гладкость функции и необходимость многократно вычислять значения функции и её градиентов.