

Минобрнауки России

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«САРАТОВСКИЙ национальный исследовательский ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»**

**ЗАДАНИЕ №5**

**Вариант 24**

по дисциплине «Методы оптимизации»

студента 3 курса 341 группы  
направления 02.03.03 Математическое обеспечение и администрирование информационных систем

факультета компьютерных наук и информационных технологий

Филиппенко Дмитрия Александровича

Саратов 2024



**Метод градиентного спуска**

import numpy as np

# Целевая функция

def objective\_function(x):

    x1, x2 = x[0], x[1]

    return 100 \* x1 \*\* 2 - 23 \* x1 \* x2 + 111 \* x2 \*\* 2 - 17 \* x1 + 13 \* x2 + 10

# Вычисление градиента

def compute\_gradient(x):

    x1, x2 = x[0], x[1]

    grad\_x1 = 200 \* x1 - 23 \* x2 - 17

    grad\_x2 = 222 \* x2 - 23 \* x1 + 13

    return np.array([grad\_x1, grad\_x2])

def adaptive\_gradient\_descent(func, grad\_func, x0, tolerance=1e-6, max\_iterations=1000, initial\_step=0.1):

    x = x0

    step\_size = initial\_step

    path = [x]

    for \_ in range(max\_iterations):

        grad = grad\_func(x)

        grad\_norm = np.linalg.norm(grad)

        if grad\_norm < tolerance: #∥∇f(x)∥<ϵ

            break

        x\_new = x - step\_size \* grad / grad\_norm #y=x−α∇f(x)

        if func(x\_new) > func(x):  # Уменьшаем шаг, если значение функции увеличивается

            step\_size \*= 0.5

        else:

            path.append(x\_new)

            if np.linalg.norm(x\_new - x) < tolerance:

                break

            x = x\_new

    return x, len(path)

# x0 = np.array([0.0, 0.0])

# gradient\_result = adaptive\_gradient\_descent(objective\_function, compute\_gradient, x0)

x0 = np.array([0.0, 0.0])  # Начальная точка

result, iterations = adaptive\_gradient\_descent(objective\_function, compute\_gradient, x0)

print(f"Минимум функции достигается в точке: {result}")

print(f"Количество итераций: {iterations}")

print(f"Значение функции в найденной точке: {objective\_function(result)}")

Минимум функции достигается в точке: **[ 0.07920912 -0.05035156]**

Количество итераций: **10**

Значение функции в найденной точке: **8.999430147538234**

Рассмотрим итерационную процедуру минимизации x ^k+1 = x^k +αk p^k   
где направление убывания p^k определяется с учетом информации о частных  
производных функции f (x) , а величина шага αk > 0 такова, что  
f (xk+1) < f (xk )   
Так как функция предполагается дифференцируемой, то в качестве  
прекращения итераций можно выбрать условие ∇f (xk ) < ε .

Положим на каждом шаге pk = −∇f (xk ). Если ∇f (xk ) ≠ 0, то  
условие (∇f (xk ), pk ) < 0 , очевидно, выполнено. Следовательно, направление  
вектора pk является направлением убывания функции f (x) , причем в малой  
окрестности точки xk направление pk обеспечивает наискорейшее убывание  
этой функции. Поэтому можно найти такое αk > 0, что будет выполняться  
условие.

**Метод наискорейшего спуска**

import numpy as np

from scipy.optimize import minimize\_scalar

**# Целевая функция**

def objective\_function(x):

    x1, x2 = x[0], x[1]

    return 100 \* x1 \*\* 2 - 23 \* x1 \* x2 + 111 \* x2 \*\* 2 - 17 \* x1 + 13 \* x2 + 10

**# Вычисление градиента**

def compute\_gradient(x):

    x1, x2 = x[0], x[1]

    grad\_x1 = 200 \* x1 - 23 \* x2 - 17

    grad\_x2 = 222 \* x2 - 23 \* x1 + 13

    return np.array([grad\_x1, grad\_x2])

**# Метод наискорейшего спуска**

def fastest\_descent(func, grad\_func, x0, tolerance=1e-6, max\_iterations=1000):

    x = x0

    path = [x]

    for \_ in range(max\_iterations):

        grad = grad\_func(x)

        # Определение оптимального шага с помощью минимизации одномерной функции

        optimal\_step = minimize\_scalar(lambda step: func(x - step \* grad)).x

        # Обновление текущей точки

        x\_new = x - optimal\_step \* grad

        path.append(x\_new)

        # Проверка условия сходимости

        if np.linalg.norm(x\_new - x) < tolerance:

            break

        x = x\_new

    return x, len(path)

x0 = np.array([0.0, 0.0])

result, iterations = fastest\_descent(objective\_function, compute\_gradient, x0)

print(f"Минимум функции достигается в точке: {result}")

print(f"Количество итераций: {iterations}")

print(f"Значение функции в найденной точке: {objective\_function(result)}")

Минимум функции достигается в точке**: [ 0.07920929 -0.05035241]**

Количество итераций: **7**

Значение функции в найденной точке: **8.99943014748755**

Данный метод является вариантом градиентного спуска. Здесь также  
полагают pk = −∇f (xk ), но величина шага αk находится в результате  
решения задачи одномерной оптимизации  
Φk (α) → min , Φk (α) = f (xk −α∇f (xk )) , α > 0 т. е. на каждой итерации в направлении антиградиента совершается  
исчерпывающий спуск.

**Метод Ньютона**

import numpy as np

**# Целевая функция**

def objective\_function(x):

    x1, x2 = x[0], x[1]

    return 100 \* x1 \*\* 2 - 23 \* x1 \* x2 + 111 \* x2 \*\* 2 - 17 \* x1 + 13 \* x2 + 10

**# Вычисление градиента**

def compute\_gradient(x):

    x1, x2 = x[0], x[1]

    grad\_x1 = 200 \* x1 - 23 \* x2 - 17

    grad\_x2 = 222 \* x2 - 23 \* x1 + 13

    return np.array([grad\_x1, grad\_x2])

**# Вычисление гессиана**

def compute\_hessian(x):

    return np.array([[264, -115],[-115, 264]])

**# Метод Ньютона**

def newton\_optimization(func, grad\_func, hessian\_func, x0, tolerance=1e-6, max\_iterations=1000):

    x = x0

    path = [x]

    for \_ in range(max\_iterations):

        grad = grad\_func(x)  # Вычисляем градиент

        hessian = hessian\_func(x)  # Вычисляем гессиан

        # Решаем систему линейных уравнений: H(x) \* Δx = -grad(x)

        delta\_x = np.linalg.solve(hessian, -grad)

        x\_new = x + delta\_x  # Обновляем точку

        path.append(x\_new)

        # Проверка сходимости

        if np.linalg.norm(x\_new - x) < tolerance:

            break

        x = x\_new

    return x, len(path)

# Начальная точка

x0 = np.array([0.0, 0.0])

newton\_result, newton\_iterations = newton\_optimization(objective\_function, compute\_gradient, compute\_hessian, x0)

print(f"Минимум функции методом Ньютона достигается в точке: {newton\_result}")

print(f"Количество итераций: {newton\_iterations}")

print(f"Значение функции в найденной точке: {objective\_function(newton\_result)}")

Минимум функции методом Ньютона достигается в точке: **[ 0.07920871 -0.05035154]**

Количество итераций: **14**

Значение функции в найденной точке: **8.99943014759571**

Начальная точка: Задаётся начальная точка x0​, точность ϵ>0, и максимальное количество итераций N.

Вычисление градиента и матрицы Гессе: В текущей точке xk ​ вычисляется градиент ∇f(xk) и матрица Гессе H(xk)

Решение системы линейных уравнений: Решается система:

H(xk)Δx=−∇f(xk),

где Δx=xk+1−xk - вектор шага.

Обновление точки:

xk+1=xk+Δx

Проверка условия сходимости: Если норма ∥Δx∥ меньше заданного порога ϵ, то алгоритм завершает работу.

Переход к следующей итерации: Если условие сходимости не выполнено, то возвращаемся к шагу 2.

Методы **градиентного спуска**, **наискорейшего спуска** и **метода Ньютона** используются для минимизации (или максимизации) функций и оптимизации, но они различаются по своей математической основе, сложности и скорости сходимости. Давайте разберем каждый из них.

**1. Метод градиентного спуска**

Метод градиентного спуска (Gradient Descent) — это итеративный метод, который ищет минимум функции, двигаясь в направлении противоположном градиенту функции. Градиент указывает направление наибольшего увеличения функции, поэтому для минимизации мы двигаемся в обратную сторону.

**Принцип работы:**

* На каждой итерации вычисляется градиент функции (вектор частных производных).
* Затем точка обновляется по формуле: xk+1​=xk​−α∇f(xk​)
* где α— это шаг, который можно выбирать фиксированным или адаптивным способом.

**Особенности:**

* Простой и понятный алгоритм.
* Выбор шага очень важен: слишком большой шаг может привести к расходимости, а слишком маленький — к медленной сходимости.
* Могут потребоваться много итераций для достижения сходимости, особенно для сложных функций.

**Применение:**

Используется в задачах, где функция имеет гладкую форму, например, в машинном обучении для минимизации ошибок моделей.

**2. Метод наискорейшего спуска (Fastest Descent)**

Метод наискорейшего спуска (или метод наискорейшего градиентного спуска) — это улучшение стандартного метода градиентного спуска. Вместо того чтобы использовать фиксированный шаг, этот метод оптимизирует шаг для каждой итерации, минимизируя целевую функцию вдоль направления антиградиента.

**Принцип работы:**

* В отличие от градиентного спуска, где шаг выбирается заранее, метод наискорейшего спуска минимизирует функцию по направлению антиградиента с использованием одномерной минимизации.
* Для каждой итерации ищется оптимальный шаг (альфа) с помощью метода минимизации вдоль направления антиградиента: αk​=argαmin​f(xk​−α∇f(xk​))
* где αk — это оптимальный шаг для текущей итерации.

**Особенности:**

* Оптимизирует шаг для каждой итерации, что делает его более гибким и потенциально более быстрым в нахождении минимума.
* Меньше чувствителен к выбору начального шага по сравнению с обычным градиентным спуском.
* Требует вычисления одномерной минимизации на каждой итерации, что может быть вычислительно дороже.

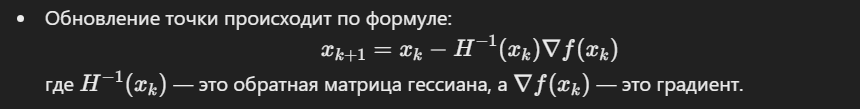
**Применение:**

Часто используется для более точной минимизации, особенно в задачах, где требуется высокая точность.

**3. Метод Ньютона**

Метод Ньютона — это более сложный и быстрый метод для поиска экстремума функции, который использует информацию о второй производной (гессиан) для ускорения сходимости.

**Принцип работы:**

* Метод Ньютона использует гессиан (матрицу вторых производных) для корректировки направления и величины шага.
* 
* Метод решает систему линейных уравнений для нахождения шага, что делает его более точным.

**Особенности:**

* Использует более точную информацию, чем метод градиентного спуска, так как учитывает не только первую, но и вторую производную.
* Быстро сходится (квадратично), если функция достаточно гладкая и гессиан обратим.
* Требует вычисления и инвертирования гессиана, что может быть вычислительно дорогим, особенно для высокоразмерных задач.

**Применение:**

Часто используется для минимизации сложных функций, где требуется высокая точность и скорость сходимости. Применяется в задаче оптимизации, где функция имеет хорошую форму и гессиан легко инвертируется.